A Separation Logic for Effect Handlers

PAULO EMÍLIO DE VILHENA, Inria, France
FRANÇOIS POTTIER, Inria, France

User-defined effects and effect handlers are advertised and advocated as a relatively easy-to-understand and modular approach to delimited control. They offer the ability of suspending and resuming a computation and allow information to be transmitted both ways between the computation, which requests a certain service, and the handler, which provides this service. Yet, a key question remains, to this day, largely unanswered: how does one modularly specify and verify programs in the presence of both user-defined effect handlers and primitive effects, such as heap-allocated mutable state? We answer this question by presenting a Separation Logic with built-in support for effect handlers, both shallow and deep. The specification of a program fragment includes a protocol that describes the effects that the program may perform as well as the replies that it can expect to receive. The logic allows local reasoning via a frame rule and a bind rule. It is based on Iris and inherits all of its advanced features, including support for higher-order functions, user-defined ghost state, and invariants. We illustrate its power via several case studies, including (1) a generic formulation of control inversion, which turns a producer that "pushes" elements towards a consumer into a producer from which one can "pull" elements on demand, and (2) a simple system for cooperative concurrency, where several threads execute concurrently, can spawn new threads, and communicate via promises.

Additional Key Words and Phrases: separation logic, effect handlers, program verification

ACM Reference Format:
Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proc. ACM Program. Lang.* 5, POPL, Article 33 (January 2021), 28 pages. https://doi.org/10.1145/3434314

1 INTRODUCTION

User-defined effects and effect handlers [Plotkin and Pretnar 2009; Kammar et al. 2013; Bauer and Pretnar 2015] offer an appealing basis for modular effectful programming. This programming language feature allows separating, on the one hand, a piece of client code that assumes the availability of certain effectful operations such as "writing to a file", "sending an element to a consumer", "yielding control to some other thread", and so on; and, on the other hand, one or more effect handlers, which provide implementations of these services.

From an operational point of view, performing an effect is very much like raising an exception: control is transferred to an enclosing handler. An effect handler resembles an exception handler, with one key difference: unlike an exception handler, it has access to a continuation. This continuation represents the computation that has been suspended by this effect and that awaits a reply from the handler. By invoking this continuation, the handler can communicate a reply to the suspended computation and resume its execution. Because they allow capturing part of the evaluation context and reifying it as a continuation, effects and effect handlers are a form of delimited control.

Authors' addresses: Paulo Emílio de Vilhena, Inria, France, paulo-emilio.de-vilhena@inria.fr; François Pottier, Inria, France, francois.pottier@inria.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2475-1421/2021/1-ART33
https://doi.org/10.1145/3434314

Effect handlers are found in several research programming languages, such as Eff [Bauer and Pretnar 2015, 2020], Frank [Lindley et al. 2017], Koka [Leijen 2014, 2020], Links [Hillerström et al. 2020], and Multicore OCaml [Dolan et al. 2017, 2020]. They have also been implemented as a library in mainstream programming languages such as Scala [Brachthäuser et al. 2020]. Several of these programming languages have primitive effects such as input/output, dynamically-allocated mutable state, and concurrency. There is a growing need for programmers and researchers to understand how to reason about the combination of primitive effects and user-defined effect handlers.

Although control operators are often regarded as difficult to understand, one may argue that this is perhaps the result of an overly mechanistic point of view. Let us draw an analogy with recursion. We believe that many programming teachers would agree that a key to understanding and mastering recursion is to avoid thinking in terms of what the machine does (jump from caller to callee and back; save and restore local data using a stack) and to instead think about what the recursive call requires and what it achieves, that is, to reason based on a *precondition* and a *postcondition*. Similarly, we argue that, in order to understand and master effect handlers, programmers should not think in terms of captured continuations and jumps. Instead, they should be able to rely on a set of simple *reasoning rules* that allow them to think *modularly* about effects. In particular, one should reason about performing an effect essentially in the same way as one reasons about calling a function, that is, in terms of a precondition and postcondition, without worrying about the mechanics of effects.

Separation Logic [Reynolds 2002; O'Hearn 2019], a descendant of Floyd-Hoare logic, has proved an extremely powerful and scalable way of reasoning about programs in the presence of complex features such as dynamically-allocated mutable state, higher-order functions, and shared-memory concurrency. We wish to extend this methodology with support for reasoning about effect handlers. Our foundation is Iris [Jung et al. 2018], a modern Separation Logic whose metatheory and user interface are embedded in Coq. Iris provides a rich setting in which we can prove the soundness of our reasoning rules and carry out a number of case studies.

One-shot versus multi-shot continuations. Among the design choices that we must make, the issue of *multi-shot* versus *one-shot* continuations is fundamental. The question is, should it be permitted or forbidden to invoke a captured continuation more than once?

Some programming languages allow a continuation to be invoked at most once. In Multicore OCaml [Dolan et al. 2017], for instance, this rule is enforced via a runtime check. One might think that this restriction is motivated by performance considerations: supporting multiple invocations of a continuation is likely to require copying stack segments, an expensive operation. In reality, though, there is a less obvious, more fundamental motivation for this restriction: *allowing continuations to be invoked more than once breaks certain fundamental laws of reasoning about programs.*

In short, if a continuation can be called twice, then a code block can be entered once and exited twice. This implies that some familiar forms of reasoning are in fact incorrect. As an example, consider the following fragment of OCaml code, where nothing is known about the function `f`:

```
1  let x = ref 0 in    (* initialize x to zero *)
2  f();                (* f has no access to x, which remains zero *)
3  x := !x + 1;        (* increment x from zero to one *)
4  assert (!x = 1);    (* x now contains one *)
```

In plain OCaml, the runtime assertion on the last line must succeed. The informal argument that appears in the comments forms the basis for a formal proof in Separation Logic. The proof is simple, and relies only on the most fundamental principles of the logic, namely the rules for allocating, reading, and writing a memory location, the rule of sequential composition (also known as the "bind rule"), and the frame rule. A proof outline goes as follows: (1) at the end of line 1, the points-to assertion $x \mapsto 0$ appears; (2) under the assumption that f satisfies the specification

$\{\mathsf{True}\}\ f\ ()\ \{\mathsf{True}\}$, the frame rule guarantees that it also satisfies $\{x \mapsto 0\}\ f\ ()\ \{x \mapsto 0\}$, so, at the end of line 2, the assertion $x \mapsto 0$ remains available; (3) therefore, at the end of line 3, we have $x \mapsto 1$, so (4) the assertion on line 4 must succeed.

In Multicore OCaml, which extends OCaml with effect handlers and one-shot continuations, it is still true that this assertion must succeed. In this setting, our aim is to propose an extension of Separation Logic where the above proof outline remains valid, regardless of which effects the function f might perform. In particular, we wish to keep the bind rule and the frame rule of Separation Logic, essentially unchanged. We show that this is possible.

In a hypothetical extension of OCaml with effect handlers and multi-shot continuations, though, the assertion on line 4 would no longer be correct. Indeed, if the function `f` performed an effect and if the captured continuation was invoked twice by an enclosing effect handler, then `x` would be incremented twice, first from 0 to 1, then from 1 to 2, causing the assertion to fail [de Vilhena and Pottier 2020c]. This shows that certain means of reasoning about program correctness are broken by the introduction of multi-shot continuations. This is not an entirely new discovery: Dreyer et al. [2012] note that call/cc breaks the "well-bracketing" of computations, while Timany and Birkedal [2019] note that it breaks the bind rule of Separation Logic. However, neither of these papers considers delimited continuations or points out that multi-shot continuations cause more breakage than one-shot continuations. We compare our work with Timany and Birkedal's in §7. Extending our system with multi-shot continuations is future work (§8).

Other Design Choices. One distinguishes "shallow" and "deep" effect handlers [Kammar et al. 2013; Hillerström and Lindley 2018]. A shallow handler serves its purpose at most once: after it has handled one effect, it disappears. A deep handler is persistent: after it has handled one effect, it remains installed (as the topmost frame of the captured continuation), so it is able to handle more effects. Shallow and deep handlers are useful in different situations. In this paper, we wish to reason about both forms of handlers. Because it is easy to encode deep handlers on top of shallow handlers, we view shallow handlers as a primitive form and deep handlers as a derived form.

Like exceptions, effects are usually named. A handler deals only with a few specific effects, designated by their names. In this paper, though, we work with only one unnamed effect. Thus, an effect is always handled by the nearest enclosing handler. This allows us to focus on the core issue of reasoning about the interaction between an effectful computation and its handler. Labeling effects and handlers with names, and dynamically allocating fresh names, is future work (§8).

Contributions. We develop a Separation Logic, based on Iris, for a call-by-value calculus featuring primitive heap-allocated mutable state and effect handlers (§2). We allow one-shot continuations only. We argue that the specification of a program fragment should include a protocol that describes the effects that the program may perform and the replies that it can expect to receive. We introduce a notion of protocol (§3) that is simple yet expressive. Our reasoning rules (§4) include the key rules of Separation Logic, including a frame rule and a bind rule that is restricted to "neutral" contexts. We prove the soundness of our logic and illustrate its expressiveness via two small yet nontrivial case studies, namely a generic implementation of control inversion (§5) and a simple implementation of cooperative concurrency (§6). Our proofs are machine-checked [de Vilhena 2020]. The current limitations of our work (§8) include the absence of shared-memory concurrency, multi-shot continuations, and multiple named effects.

2 SYNTAX AND SEMANTICS OF *HH*

We present *HH*, for "heaps and handlers", a call-by-value λ-calculus equipped with primitive mutable state and effect handlers. Its syntax and operational semantics are mostly standard: a reader who is familiar with effect handlers may skim through this section. A slightly unusual feature of our

$$\begin{array}{rl} v, w, k, h, r ::= & \mu f.\lambda x.\, e \mid () \mid (v, v) \mid \mathsf{inj}_i\ v \mid \ell \\ & \mid (\lambda N)^\ell \\ e ::= & x \mid \mu f.\lambda x.\, e \mid e\ e \\ & \mid () \mid (e, e) \mid \mathsf{proj}_i\ e \\ & \mid \mathsf{inj}_i\ e \mid (\mathsf{match}\ e\ \mathsf{with}\ v \mid v) \\ & \mid \ell \mid \mathsf{ref}\ e \mid !\,e \mid e := e \\ & \mid \mathsf{do}\ e \mid \S(N)[\mathsf{do}\ v] \\ & \mid (\mathsf{shallow\text{-}try}\ e\ \mathsf{with}\ v \mid v) \\ & \mid (\lambda N)^\ell \end{array} \qquad \begin{array}{rl} K ::= & [] \\ & \mid (\mathsf{shallow\text{-}try}\ K\ \mathsf{with}\ v \mid v) \\ & \mid e\ K \mid K\ v \mid (e, K) \mid (K, v) \mid \mathsf{proj}_i\ K \\ & \mid \mathsf{inj}_i\ K \mid (\mathsf{match}\ K\ \mathsf{with}\ v \mid v) \\ & \mid \mathsf{ref}\ K \mid !\,K \mid e := K \mid K := v \mid \mathsf{do}\ K \\ N ::= & [] \\ & \mid e\ N \mid N\ v \mid (e, N) \mid (N, v) \mid \mathsf{proj}_i\ N \\ & \mid \mathsf{inj}_i\ N \mid (\mathsf{match}\ N\ \mathsf{with}\ v \mid v) \\ & \mid \mathsf{ref}\ N \mid !\,N \mid e := N \mid N := v \mid \mathsf{do}\ N \end{array}$$

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

semantics is that it includes a "one-shot check", a runtime check that causes a crash if a continuation is invoked twice. With respect to this semantics, we prove that our program logic is sound: a program that is verified using our logic cannot crash (Theorem 4.1). This proves that our logic enforces an affine usage of continuations. Naturally, the logic is also sound with respect to the simpler semantics that would be obtained by removing the one-shot check.

2.1 Syntax

The syntax of *HH* appears in Figure 1. Its features include recursive functions, binary products and sums, mutable references, and constructs for performing and handling effects. There is no "let" construct: sequential composition is encoded as a β-redex.

The construct "do e" first evaluates e to a value v, then performs an effect. This is analogous to raising an exception: when this construct is executed, evaluation is suspended, and control is transferred to the nearest enclosing handler. Whereas raising an exception aborts evaluation, performing an effect suspends it: the current evaluation context N, up to (and excluding) the nearest enclosing handler, is captured and turned into a first-class continuation $(\lambda N)^\ell$. The memory location ℓ, which is allocated fresh, stores a Boolean flag, and is initialized with true. It plays a role in the one-shot check. The handler receives both the value v and the continuation as arguments. Thus, the value v is transmitted from the computation to the handler.

The construct $\S(N)[\mathsf{do}\ v]$ is an active effect, that is, a do construct that has captured the partial evaluation context N, and can possibly capture a larger context still. It plays a role in the small-step operational semantics and is not accessible to the programmer. In the literature, it is often written $\mathsf{op}\ v\ N$ [Plotkin and Pretnar 2009; Kammar et al. 2013].

The construct "shallow-try e with $h \mid r$" wraps the expression e in a shallow effect handler, which consists of two branches: the function h handles effects, whereas the function r handles normal termination. The metavariables h and r range over values.

2.2 Encoding Deep Handlers

The construct "deep-try e with $h \mid r$", which installs a deep handler, is sugar for the function application $deep\ (\lambda().\, e)\ h\ r$, where the recursive function $deep$ is defined as follows:

$$deep \triangleq \mu deep.\lambda e\, h\, r.\ \mathsf{shallow\text{-}try}\ e()\ \mathsf{with}\ \lambda v\, k.\ h\ v\ (\lambda x.\ deep\ (\lambda().\ k\ x)\ h\ r) \mid r$$

Instead of passing to the handler the continuation k, we pass it $\lambda x.\ deep\ (\lambda().\ k\ x)\ h\ r$, where the call $k\ x$ is wrapped in a new instance of the effect handler. Thus, a deep handler reinstalls itself as the top frame of the continuation. This encoding is standard [Hillerström and Lindley 2018, §3.1].

$$
\begin{array}{lrcl}
\text{(R1)} & \mathsf{do}\ v\ /\ \sigma & \longrightarrow & \S([\,])[\mathsf{do}\ v]\ /\ \sigma \\
\text{(R2)} & e_1\ (\S(N)[\mathsf{do}\ v_2])\ /\ \sigma & \longrightarrow & \S(e_1\ N)[\mathsf{do}\ v_2]\ /\ \sigma \\
\text{(R3)} & (\S(N)[\mathsf{do}\ v_1])\ v_2\ /\ \sigma & \longrightarrow & \S(N\ v_2)[\mathsf{do}\ v_1]\ /\ \sigma \\
\text{(R4)} & \mathsf{shallow\text{-}try}\ \S(N)[\mathsf{do}\ v]\ \mathsf{with}\ h \mid r\ /\ \sigma & \longrightarrow & h\ v\ (\lambda N)^{\ell}\ /\ \sigma[\ell \mapsto \mathsf{true}] \\
\text{(R5)} & \mathsf{shallow\text{-}try}\ v\ \mathsf{with}\ h \mid r\ /\ \sigma & \longrightarrow & r\ v\ /\ \sigma \\
\text{(R6)} & (\lambda N)^{\ell}\ v\ /\ \sigma[\ell \mapsto \mathsf{true}] & \longrightarrow & N[v]\ /\ \sigma[\ell \mapsto \mathsf{false}]
\end{array}
$$

Fig. 2. Head Reduction (Selected Rules)

2.3 Semantics

The small-step operational semantics of *HH* is inspired by Kammar et al.'s semantics [2013], yet is slightly simpler, as we do not have named effects. Because *HH* has primitive mutable state, its operational semantics involves stores σ, which are finite maps of memory locations to values. Some of the rules that define the head reduction relation $e\ /\ \sigma \to e'\ /\ \sigma'$ appear in Figure 2.

Rule R1 transforms "do v", a passive expression, into "$\S([\,])[\mathsf{do}\ v]$", an active effect that is in the process of capturing its evaluation context. At the beginning, it has captured the empty context $[\,]$. The following rules allow it to progressively capture a larger context, one frame at a time.

Rules R2 and R3 allow the construct $\S(N)[\mathsf{do}\ e]$ to capture one frame of the evaluation context that surrounds it. There are more rules in this style (not shown). These rules allow an active effect to capture all of its evaluation context, step by step, up to either the nearest enclosing handler or the top level. In the latter case, it becomes stuck; this is a runtime error, an unhandled effect. (Our program logic rules out all runtime errors, including this one.) In the former case, one of the next two reduction rules applies.

Rules R4 and R5 define the behavior of the "shallow-try" construct. If an effect is performed, the first branch of the handler takes control and receives the value v and the continuation $(\lambda N)^{\ell}$ as arguments. If a value v is returned, the second branch takes control and receives v as an argument.

Rule R6 defines the behavior of a first-class continuation $(\lambda N)^{\ell}$. Provided ℓ contains true, an application of this continuation to a value w reduces to $N[w]$, which means that the suspended computation is resumed, just as if the expression "do v" had returned the value w. Furthermore, ℓ is updated with false, so another attempt to apply this continuation will be stuck. This is the one-shot check.

The "neutral" evaluation contexts N are defined in Figure 1. They *do not* include the form "shallow-try $[\,]$ with $h \mid r$": such a frame is never captured inside a continuation. In this calculus, the evaluation context is always captured up to the nearest enclosing handler, and no further.

The unrestricted evaluation contexts K *do* include the form "shallow-try $[\,]$ with $h \mid r$". They are used in the definition of the reduction relation, which states that if the head reduction step $e\ /\ \sigma \to e'\ /\ \sigma'$ is permitted, then the reduction step $K[e]\ /\ \sigma \longrightarrow K[e']\ /\ \sigma'$ is permitted. This allows reduction to take place under a handler.

3 PROTOCOLS AND SPECIFICATIONS

Before inventing reasoning rules for effects and handlers, we must determine under what form we expect to write specifications. A specification describes the interaction between a program fragment (an expression) and its environment (the context in which this expression appears). We refer to the participants of this dialogue as Player and Opponent. A specification should describe the behavior of Player, insofar as this behavior can be observed by Opponent.

In the traditional setting of an imperative programming language, without effects and handlers, the interaction between an expression and its environment is relatively limited. An expression either diverges (an unobservable outcome) or returns a value; furthermore, it may modify the global state. In such a setting, a specification for an expression e can be expressed as a Hoare triple $\{P\}\ e\ \{\Phi\}$. The precondition P, an assertion, is a requirement on the initial state. The postcondition Φ, a function of a value to an assertion, is a guarantee about the final state. Under a partial correctness interpretation, which we adopt in this paper, the triple $\{P\}\ e\ \{\Phi\}$ means roughly that if the initial state satisfies P, then it is safe to execute e, and this execution either diverges or eventually returns a value v such that the final state satisfies $\Phi(v)$.

Once effects are introduced, a new kind of observable behavior appears: an expression may diverge, return a value, or perform an effect. Divergence remains unobservable; thus, we expect a specification to describe which assertions about the global state hold in each of the last two cases. Furthermore, in the last case, where the expression performs an effect, the execution of the expression is suspended (captured in a continuation) and can be resumed by the environment, if it so chooses, by invoking the continuation. Thus, the dialogue between Player and Opponent is not necessarily over: it may continue, either immediately or at some later time. We believe that it makes sense for a specification to describe this dialogue in its entirety. Therefore, a specification should include a *protocol*, a description of the requests that Player may emit (by performing effects) and of the replies that Opponent may provide (by handling these effects). This is akin to the manner in which two processes might communicate over a channel while obeying a predetermined protocol. In the following, we introduce protocols (§3.1), give examples of protocols (§3.2), characterize the exact meaning of protocols (§3.3), and give a concrete implementation of protocols in Coq (§3.4). Then, we come back to the manner in which protocols appear in specifications (§3.5).

3.1 Protocols Steps and Protocols

The interaction between an expression and its environment can be viewed as a sequence of *requests* by the expression and *replies* by the environment. An expression sends a request via the construct "do v", where the value v describes what service is requested. The environment (that is, the nearest enclosing effect handler) receives the value v, as well as a continuation k, and may send a reply via the continuation invocation $k\ w$, where w is a value. Then, the execution of the expression is resumed, and the dialogue continues. The manner in which it continues may depend on the request v and reply w that have been exchanged. For a moment, though, let us put aside the remainder of the dialogue and focus on describing one exchange of a request and a reply. We introduce the notion of a *protocol step* Ψ, a specification of what request and reply may be sent. Protocols, we shall see, can be viewed as repetitions of protocol steps.

3.1.1 Protocol Steps. What should a protocol step Ψ look like? Three points come to mind. First, a protocol step should at least specify what request v may be sent by Player and what reply w may be sent by Opponent. Second, because a request and a reply may convey information about the current state, and may be used to transfer the ownership of a resource, a protocol step should include a precondition (an assertion P) and a postcondition (an assertion Q). In Separation Logic, an assertion carries both information about and access rights for a piece of physical or ghost state. Third and last, because it is a *specification* of an exchange, a protocol step should be nondeterministic: that is, it should allow Player to choose within a range of permitted requests, and should allow Opponent to choose within a range of permitted replies. This flexibility can be obtained by letting a protocol step include binders $\vec{x}$ and $\vec{y}$ (where the number of variables and their types is up to the user). With these points in mind, we propose the following way of constructing a protocol step:

$$\Psi \mathrel{+}= \ !\ \vec{x}\ (v)\ \{P\}.\ ?\ \vec{y}\ (w)\ \{Q\} \qquad \text{(one way of constructing a protocol step)}$$

The scope of the binders $\vec{x}$ and $\vec{y}$ extends all the way towards the right. The intuitive meaning of this protocol step is as follows. When making a request, Player may choose any instance of the variables $\vec{x}$, provided the assertion P holds. The ownership of the resources governed by P is then transferred from Player to Opponent. Symmetrically, when replying, Opponent may choose any instance of the variables $\vec{y}$, provided Q holds. The ownership of the resources governed by Q is transferred from Opponent to Player.

It may occur to the reader that this is very much like a polymorphic specification for a function. This is true: indeed, we would like programmers to think about performing an effect essentially in the same way as they think about calling a function.

Although not absolutely necessary, we find it convenient to have a few more ways of constructing protocols steps:

$$\Psi \ \mathrel{+}= \ \bot \mid \Psi + \Psi \mid f \mathbin{\#} \Psi \qquad\qquad \text{(three more ways of constructing protocol steps)}$$

The intuitive interpretation of these constructs is as follows. The protocol step $\Psi_1 + \Psi_2$ allows Player to choose between Ψ_1 and Ψ_2. The empty protocol step $\bot$, which does not allow Player to perform any effect, serves as the unit of this binary choice operator. Finally, the meaning of the protocol step $f \mathbin{\#} \Psi$, where f is an injective function of values to values, is this: if Ψ allows Player to make the request v, then $f \mathbin{\#} \Psi$ allows Player to make the request $f(v)$. This is typically used to mark a value with a tag: the protocol *Coop* in Figure 12 offers an example.

This concludes our presentation of how protocol steps can be constructed. We avoid speaking of the "syntax" of protocol steps: instead, we present the type of protocol steps to a user of our logic as an abstract type, equipped with the four constructors shown above. This abstract type is equipped with one more public operation, namely an interpretation, whose presentation we defer to §3.3.

3.1.2 Protocols. A dialogue between Player and Opponent is a *sequence* of requests and replies. A protocol is a specification of such a dialogue: it prescribes which sequences of requests and replies are permitted. What should a protocol look like? One might imagine that a protocol should be a (finite or infinite) sequence of protocol steps. One might also imagine that, because the choices made by the two participants during the first exchange can influence the remainder of the dialogue, the scope of the binders $\vec{x}$ and $\vec{y}$ that appear in the first step of the protocol should extend to include the remainder of the protocol. That would give rise to a notion of protocol akin to the *dependent separation protocols* found in Actris [Hinrichsen et al. 2020]. In the early stages of this work, we did in fact follow such a route. However, we later discovered that one can settle for much simpler protocols: we propose to work with protocols of the form Ψ, whose intended meaning is that the protocol step Ψ can be repeated an arbitrary number of times.[1] In short, *a protocol is a repetition* of a protocol step.

Why is it acceptable to work with such simplistic protocols? It may seem at first that, if every protocol must be a repetition of a fixed protocol step Ψ, then there is no way of keeping track of the *current state* of the protocol, that is, of letting the messages that are exchanged in the beginning influence what messages are permitted next. In reality, though, because a protocol step Ψ can contain preconditions and postconditions, it does allow keeping track of a state. By choosing appropriate pre- and postconditions, one can express the idea that performing an effect causes a state change, and one can allow certain effects to be performed only in certain states. This is illustrated by some of the examples that follow.

[1] Although we could in principle introduce distinct abstract types and use distinct metavariables for protocol steps and protocols, we simply work with protocol steps everywhere. Thus, when we speak of "the protocol Ψ", we mean the protocol that allows repeating the protocol step Ψ as many times as one wishes.

3.2 Examples of Protocols

3.2.1 Abort. Let us begin with an extremely simple example. Suppose that one wishes to define an operation "*abort*" whose effect is to abort the current computation and irreversibly transfer control to an enclosing handler, just like an exception with no argument. *abort* can be viewed as sugar for the expression "do ()", which performs an effect whose argument is the unit value (). A programmer's assumption, when using *abort*, is that the handler never invokes the captured continuation. This guarantees that one can regard *abort* as an expression that never returns. This contract is expressed by the following protocol:

$$ABORT \triangleq !\ (())\ \{\mathsf{True}\}.\ ?\, y\ (y)\ \{\mathsf{False}\}$$

This protocol begins with the value () and the precondition True, which means that it is permitted to use *abort* at any time. Then, the protocol requires the handler to reply with some value y that satisfies False. Because False is unsatisfiable, the handler is in fact not allowed to reply: it must not invoke the captured continuation. In this example, the repetition that is implicit in every protocol plays no role: as soon as one effect is performed, the dialogue is over.

3.2.2 Memory Cell with Exchange. Suppose that there exists a mutable memory cell at location ℓ in the heap, and suppose that, instead of offering direct access to this location, one wishes to install a handler for an "exchange" effect. This handler writes a new value into the memory cell and returns its previous value. Because the heap is updated, the protocol that describes this effect must involve nontrivial pre- and postconditions.

The simplest thing to do is to let a "points-to" assertion appear in the pre- and postcondition. This leads to the following protocol:

$$CXCHG \triangleq !\, x\, x'\ (x')\ \{\ell \mapsto x\}.\ ?\ (x)\ \{\ell \mapsto x'\}$$

This protocol binds two variables x and x', which can be intuitively regarded as universally quantified: indeed, when performing an "exchange" effect, Player can instantiate x and x' with two arbitrary values v and v'. Thus, this protocol states that, provided the location ℓ currently contains some value v, Player may perform the effect "do v'". If Opponent (that is, the handler) wishes to reply, then it must update the memory location ℓ with the value v' and reply by returning the value v. During this interaction, the unique ownership of the memory location is transferred from Player to Opponent and back. In this example, the repetition that is implicit in every protocol is exploited: this protocol allows performing as many "exchange" effects as one wishes.

Slightly more elegantly, while still using the above protocol, one can encapsulate the concrete assertion "$\ell \mapsto x$", in Player's eyes, as an abstract predicate $I\ x$. From Player's perspective, the protocol is then:

$$AXCHG \triangleq !\, x\, x'\ (x')\ \{I\ x\}.\ ?\ (x)\ \{I\ x'\}$$

From Opponent's perspective, the protocol remains *CXCHG*, so the handler has access to the memory cell at address ℓ. Player, though, is unaware of the manner in which the "exchange" effect is implemented, and cannot read or write the memory cell except via this effect.

This example shows that, even though this protocol is a repetition of a fixed protocol step, it does keep track of a "current state", which in this case is just the value currently stored in the memory cell, and it does so without revealing how this state is actually represented in memory.

3.2.3 Memory Cell with Read and Write. The previous example uses a single "exchange" effect to perform a combination of a read and a write. One can also separate these operations. Let us write *read* for the left injection inj_1 and *write* for the right injection inj_2. We adopt the convention that a request must be either *read*(), in which case Opponent must reply with the content of the memory cell, or *write* v', in which case it must update the memory cell and reply with the value (). This is described by the following protocol, where an abstract predicate I represents the current state:

$$\begin{array}{rl} \mathit{READWRITE} \triangleq & \mathit{read} \,\#\, !\, x\, (())\, \{I\ x\}.\, ?\, (x)\, \{I\ x\} \\ & +\ \mathit{write} \,\#\, !\, x\, x'\, (x')\, \{I\ x\}.\, ?\, (())\, \{I\ x'\} \end{array}$$

The "choice" and "tag" constructors introduced earlier (§3.1.1) clearly show that Player has a choice between two effects and that the corresponding requests carry distinct tags.

3.2.4 Sequence of Elements. Suppose one wishes to allow Player to produce a (finite or infinite) sequence of elements, while Opponent is in charge of consuming these elements in some way. An element x is produced by performing a "yield" effect "do x". This effect is intended to be repeated as many times as there are elements in the sequence.

A protocol for this scenario must somehow keep track of the logical state of the enumeration: this ensures that the participants agree on which elements may or must still be produced. Filliâtre and Pereira [2016], followed by Pottier [2017], propose a systematic way of doing this. They describe the set of all legal (finite or infinite) sequences via two predicates *permitted* and *complete* whose argument is a list. The idea is, *permitted xs* holds if there exists at least one legal (finite or infinite) sequence of which *xs* is a prefix, while *complete xs* holds if the finite sequence *xs* is legal. Then, they represent the current state of the protocol by the list *xs* of the elements that have been produced. In this approach, the "yield" effect can be described by the protocol *SEQ I*. Here, *SEQ* is parameterized with a predicate I, which is itself parameterized with *xs*.

$$\mathit{SEQ}\ I \triangleq\ !\ \mathit{xs}\, x\ (x)\ \{\mathit{permitted}\,(\mathit{xs} \mathbin{+\!\!+} [x])\ *\ I\ \mathit{xs}\}.\ ?\ (())\ \{I\ (\mathit{xs} \mathbin{+\!\!+} [x])\}$$

The predicate I keeps track of the current state of the enumeration. Whereas $I\ \mathit{xs}$ appears in the precondition of the "yield" effect, $I\ (\mathit{xs} \mathbin{+\!\!+} [x])$ appears in its postcondition. Anticipating on §3.5, where we explain the form of a specification, let us say that a correct producer is expected to satisfy a specification of the following form:

$$\forall I.\, I\ []\ \mathrel{-\!\!*}\ \mathit{ewp}\ \mathit{producer}\ \langle \mathit{SEQ}\ I\rangle\{().\, \exists \mathit{xs}. I\ \mathit{xs}\ *\ \mathit{complete}\, \mathit{xs}\}$$

From the initial state $I\ []$, this specification requires the producer to reach a final state of the form $I\ \mathit{xs}$, where *xs* is complete. To do so, the producer may perform "yield" effects, as per the protocol *SEQ I*. Because the universal quantification over I forces the producer to view I as affine and abstract, the producer has no other way of reaching the prescribed final state than to perform "yield" effects. Because the precondition of "yield" includes $\mathit{permitted}\,(\mathit{xs} \mathbin{+\!\!+} [x])$, the producer cannot yield an element x that is illegal in light of the elements that have been produced earlier. Because the postcondition of the producer includes *complete xs*, the producer cannot terminate unless a complete list of elements has been produced.

The predicate $I\ \mathit{xs}$ encodes a logical state: it does not necessarily correspond to a piece of physical mutable state. The above definition of *SEQ* is parameterized with I, so different choices can be made in different situations. In the proof of `invert` (§5), we instantiate *SEQ* with a concrete choice of I, where $I\ \mathit{xs}$ is actually a piece of ghost state.

3.3 Interpretation and Subsumption of Protocols

We have presented an abstract type of protocol steps, equipped with four constructors (§3.1.1). It also serves as the type of protocols (§3.1.2). To complete the presentation of the public interface of this abstract type, there remains to say that this abstract type is equipped with an interpretation, which allows us to reason about what a protocol *means*.

The interpretation of protocols is just a formal version of the intuitive interpretation that was given earlier (§3.1.1). It is a 3-place predicate Ψ *allows* do v $\{\Phi\}$. Its intuitive reading is two-fold.

$$
\begin{array}{lrcl}
\text{(A1)} & !\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\}\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\} & \dashv\vdash & \exists\vec{x}.\ v' = v\ *\ P\ *\ \forall\vec{y}.\ Q \mathrel{-\!\!*} \Phi(w) \\
\text{(A2)} & \bot\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\} & \dashv\vdash & \mathsf{False} \\
\text{(A3)} & \Psi_1 + \Psi_2\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\} & \dashv\vdash & \Psi_1\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\}\ \vee \\
 & & & \Psi_2\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\} \\
\text{(A4)} & f \mathbin{\#} \Psi\ \mathit{allows}\ \mathsf{do}\ f(v')\ \{\Phi\} & \dashv\vdash & \Psi\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi\} \\
\text{(A5)} & \Psi\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi_1\}\ *\ \forall w.\, \Phi_1(w) \mathrel{-\!\!*} \Phi_2(w) & \vdash & \Psi\ \mathit{allows}\ \mathsf{do}\ v'\ \{\Phi_2\}
\end{array}
$$

Fig. 3. Properties of the predicate Ψ *allows* do v $\{\Phi\}$.

First, the protocol Ψ allows making the request v. Furthermore, the protocol Ψ guarantees that every permitted reply w satisfies $\Phi(w)$.

The assertion Ψ *allows* do v $\{w.\ \ldots\}$ is a proof obligation that Player faces when he wishes to perform an effect. (For brevity, we write $w.$ instead of $\lambda w.$) It involves a proof obligation about v: Player must prove that the request is permitted by the protocol. Furthermore, it involves a universal quantification over w: Player must prove that he is prepared to accept every permitted reply.

The interpretation of protocols satisfies the laws presented in Figure 3. The most central law, A1, defines the meaning of the protocol $!\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\}$. When performing an effect, Player chooses how to instantiate the auxiliary variables $\vec{x}$: that is, Player chooses *some* $\vec{x}$. Player must then make the request v and relinquish the resource P. If Opponent replies, then it chooses how to instantiate the auxiliary variables $\vec{y}$, which is why Player must be prepared to accept *every* choice of $\vec{y}$. Opponent must then make the reply w and relinquish the resource Q, which is why Player may assume that Q holds and must prove that Φ holds of w.

One subtle aspect must be emphasized: this law does *not* involve the persistence modality $\Box$ [Jung et al. 2018, §5.3]. This means that Player, who must establish the implication $\forall\vec{y}.Q \mathrel{-\!\!*} \Phi(w)$, can assume that this implication will be exploited at most once. Dually, Opponent, who is allowed to exploit this implication, may exploit it at most once. This will have the consequence that a captured continuation can be invoked at most once. Thus, this is where we articulate our decision to forbid multi-shot continuations. In return, we will be able to establish a standard frame rule (§4.2.4). If a persistence modality was inserted in front of the implication $\forall\vec{y}.Q \mathrel{-\!\!*} \Phi(w)$ in law A1, then this frame rule would not hold.

The remaining laws in Figure 3 are simple. Law A2 indicates that the empty protocol $\bot$ allows no effect. Law A3 indicates that the protocol $\Psi_1 + \Psi_2$ allows Player to choose between the effects permitted by Ψ_1 and those permitted by Ψ_2. Law A4 states that the protocol $f \mathbin{\#} \Psi$ allows the same requests as the protocol Ψ, but requires them to be marked with f, a function of values to values. Law A5 indicates that the assertion Ψ *allows* do v $\{\Phi\}$ is covariant in Φ.

From the interpretation of protocols, one can deduce a subsumption relation, that is, a preorder $\sqsubseteq$ on protocols. The assertion $\Psi_1 \sqsubseteq \Psi_2$ means that Ψ_1 is less permissive, from Player's point of view, than Ψ_2. It is a short-hand for $\Box(\forall v, \Phi.\ \Psi_1\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\} \mathrel{-\!\!*} \Psi_2\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\})$. The reason why a persistence modality appears in this definition is that we must not just compare two protocol *steps* Ψ_1 and Ψ_2: we must compare two *repetitions* of these steps (§3.1.2), so it must repeatedly be the case that Ψ_1 is less permissive than Ψ_2. Subsumption $\sqsubseteq$ satisfies a number of natural properties: $\bot$ is the least element; $+$ is associative, commutative, and idempotent; $\cdot + \cdot$ and $f \mathbin{\#} \cdot$ are covariant. Subsumption can be exploited via the reasoning rule Monotonicity (§4.2.4).

$$iEff \triangleq Val \to (Val \to iProp) \to iProp$$

$$\begin{aligned} !\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\} &\triangleq \lambda v'\,\Phi'.\,\exists\vec{x}.\,v' = v \ * \ P \ * \ \forall\vec{y}.\,Q \mathrel{-\!\!*} \Phi'(w) \\ \bot &\triangleq \lambda v'\,\Phi'.\,\mathsf{false} \\ \Psi_1 + \Psi_2 &\triangleq \lambda v'\,\Phi'.\,\Psi_1\,v'\,\Phi' \vee \Psi_2\,v'\,\Phi' \\ f \,\#\, \Psi &\triangleq \lambda v'\,\Phi'.\,\exists w'.v' = f(w') \ * \ \Psi\,w'\,\Phi' \end{aligned}$$

$$\Psi \ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\} \triangleq \exists\Phi'.\,\Psi\,v\,\Phi' \ * \ \forall w.\,\Phi'(w) \mathrel{-\!\!*} \Phi(w)$$

Fig. 4. Concrete implementation of protocols

3.4 Concrete Implementation of Protocols

The abstract type of protocol steps is equipped with a number of constructors (§3.1.1) and with an interpretation (§3.3) that satisfies the laws of Figure 3. That is all we need: a reader who is willing to trust that such a type exists may safely skip ahead to the next section (§3.5), which explains how we use protocols while reasoning about programs. For the reader who is curious of the low-level details, in this section, we explain how this type is defined.

The definition of the type *iEff* of protocol steps appears in Figure 4. It can be understood as follows. We wish to equip the type *iEff* with four introduction operations and with just one elimination operation, namely interpretation, whose type is $iEff \to Val \to (Val \to iProp) \to iProp$.[2] The most direct way of defining *iEff*, then, is to define it by $iEff \triangleq Val \to (Val \to iProp) \to iProp$, and to define interpretation as the identity function, that is, to pose $\Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\} \triangleq \Psi\,v\,\Phi$. This is essentially what is done in Figure 4. The type *iEff* and its four constructors are defined in such a way that, if $\Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\}$ were defined as $\Psi\,v\,\Phi$, then the laws A1 to A4 in Figure 3 would hold by definition. Defining interpretation as the identity does not quite work, though: it does not validate law A5. Fortunately, one can force a covariant interpretation of protocols by building an upward closure into the definition of interpretation; this is done in the last line of Figure 4. All of the laws in Figure 3 are then satisfied.

3.5 Protocols in Specifications

In ordinary Separation Logic, a specification is a triple $\{P\}\ e\ \{\Phi\}$, where the precondition P, an assertion, describes the initial state, and the postcondition Φ, a function of a value to an assertion, describes the final state. In Iris [Jung et al. 2018, §6], such a triple is sugar for $\Box(P \mathrel{-\!\!*} \mathit{wp}\ e\ \{\Phi\})$, where *wp* is the *weakest precondition* predicate. Jung et al. convincingly argue that working with magic wands and *wp* assertions is simpler and more powerful than working at the level of triples.

The use of the persistence modality $\Box$ in the definition of triples is a convention. It is motivated by the fact that, most of the time, one wishes to reason about unrestricted functions, which can be invoked as many times as one desires. In this paper, though, we also need to reason about one-shot continuations (§1). This gives us yet more reason to work at the level of magic wands and *wp* assertions. Because an implication such as $P \mathrel{-\!\!*} \mathit{wp}\ k()\ \{\Phi\}$ can be used at most once, it naturally expresses the fact that the function k can be invoked at most once.

We now come back to the opening question of this section: in a programming language equipped with effects and effect handlers, what form do we expect a specification to take? We have partly answered this question by introducing the concept of a protocol, which describes a set of permitted interactions between an expression and its environment. There remains to decide exactly in what

[2] In the assertion $\Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\}$, the value v has type *Val*, while the postcondition Φ has type $Val \to iProp$, where *iProp* is the type of Iris propositions.

$$
\begin{array}{lrcl}
\text{(EWP1)} & \mathit{ewp}_{\mathcal{E}}\ v\ \langle\Psi\rangle\{\Phi\} & \triangleq & {}^{\mathcal{E}}{\models\!\Rrightarrow}^{\mathcal{E}}\ \Phi(v) \\
\text{(EWP2)} & \mathit{ewp}_{\mathcal{E}}\ \S(N)[\mathsf{do}\ v]\ \langle\Psi\rangle\{\Phi\} & \triangleq & \Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{w.\ \triangleright\, \mathit{ewp}_{\mathcal{E}}\ N[w]\ \langle\Psi\rangle\{\Phi\}\} \\
\text{(EWP3)} & \mathit{ewp}_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} & \triangleq & \forall\sigma.\, S(\sigma)\ {}^{\mathcal{E}}{\Rrightarrow\!\!*}^{\emptyset} \begin{cases} e \,/\, \sigma \longrightarrow \_ \,/\, \_ \ * \\ \forall e', \sigma'.\ e \,/\, \sigma \longrightarrow e' \,/\, \sigma'\ {}^{\emptyset}{\Rrightarrow\!\!*}^{\emptyset}\ \triangleright\ {}^{\emptyset}{\models\!\Rrightarrow}^{\mathcal{E}} \\ \qquad S(\sigma') * \mathit{ewp}_{\mathcal{E}}\ e'\ \langle\Psi\rangle\{\Phi\} \end{cases} \\
& \textit{(e not a value nor an effect)} & &
\end{array}
$$

Fig. 5. Definition of the weakest-precondition predicate $\mathit{ewp}_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\}$

way protocols should appear in specifications. The simplest imaginable approach is to extend Iris's *wp* assertion with one extra parameter, a protocol Ψ, which indicates what requests may be sent and what replies may be received by the Player. So, we want to define a weakest-precondition predicate of the form $\mathit{ewp}_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\}$.[3] We name it *ewp* so as to better distinguish it from the standard *wp*.

4 A SEPARATION LOGIC FOR *HH*

We now define a Separation Logic for *HH*. In keeping with the Iris tradition [Jung et al. 2018], we follow a semantic approach: first, we give direct meaning to the assertion $\mathit{ewp}_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\}$ (§4.1). Then, independently, we establish a number of reasoning rules that can be used to prove *ewp* assertions (§4.2), and we prove that the logic is sound (§4.3).

4.1 Specifications and Their Meaning

The definition of *ewp* appears in Figure 5. It is presented as a set of three defining laws. EWP1 covers the case where the expression e is a value. EWP2 is applicable when e is an effect, that is, an expression of the form $\S(N)[\mathsf{do}\ v]$. EWP3 covers the remaining cases.

The "later" modalities $\triangleright$ that appear in EWP2 and EWP3 guard the occurrences of *ewp* on the right-hand side and thereby ensure that this recursive definition is accepted (that is, these equations do have a solution). They can be ignored by most readers; we do not explain them further.

The law EWP1 defines what it means for a trivial computation, which simply returns a value v, to satisfy protocol Ψ and postcondition Φ. As one might expect, it merely requires v to satisfy Φ. The protocol Ψ is irrelevant. Indeed, a protocol describes which effects an expression e *may* perform; there is no obligation to perform an effect.

The law EWP2 defines the meaning of the assertion $\mathit{ewp}_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\}$ when e is an active effect, that is, a construct of the form $\S(N)[\mathsf{do}\ v]$, where v is the value that is passed from the expression to the handler, and N is the captured evaluation context. This law relies on the protocol interpretation predicate Ψ *allows* do v $\{\dots\}$ (§3.3). It expresses two requirements. First, it must be the case that, according to the protocol Ψ, the request v is permitted. Second, it must be the case that, for every reply w that the protocol Ψ permits, plugging the value w into the evaluation context N yields a term $N[w]$ that behaves in accordance with the protocol Ψ and the postcondition Φ.

According to this law, after performing one effect that conforms to the protocol step Ψ, the program must still conform to the protocol Ψ. This reflects our convention that every protocol is a repetition of a protocol step (§3.1.2).

[3] The parameter $\mathcal{E}$ is a mask. Masks and mask-changing updates are part of Iris's invariant machinery [Jung et al. 2018, §2.2]. Iris's weakest-precondition assertion $wp_{\mathcal{E}}\ e\ \{\Phi\}$ is in fact parameterized with a mask, which records the currently-enabled invariants. When the mask parameter is omitted, the full mask is intended. We keep this machinery and let it appear in our definitions and reasoning rules (Figures 5, 6, 7). Because it is essentially unrelated to the focus of our paper, we do not explain it. A casual reader can understand a mask-changing update-and-implication ${}^{\mathcal{E}_1}{\Rrightarrow\!\!*}^{\mathcal{E}_2}$ as an implication $-\!\!*$.

$$\textsc{Value}\quad \frac{\Phi(v)}{ewp_{\mathcal{E}}\ v\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{Do}\quad \frac{\Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\}}{ewp_{\mathcal{E}}\ (\mathsf{do}\ v)\ \langle\Psi\rangle\{\Phi\}}$$

$$\textsc{Monotonicity}\quad \frac{ewp_{\mathcal{E}_1}\ e\ \langle\Psi_1\rangle\{\Phi_1\} \qquad \mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad \Psi_1 \sqsubseteq \Psi_2 \qquad \forall v.\, \Phi_1(v) \mathrel{-\!\!*} \Phi_2(v)}{ewp_{\mathcal{E}_2}\ e\ \langle\Psi_2\rangle\{\Phi_2\}}$$

$$\textsc{Bind}\quad \frac{ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{v.\, ewp_{\mathcal{E}}\ N[v]\ \langle\Psi\rangle\{\Phi\}\}}{ewp_{\mathcal{E}}\ N[e]\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{Bind-Pure}\quad \frac{ewp_{\mathcal{E}}\ e\ \langle\bot\rangle\{v.\, ewp_{\mathcal{E}}\ K[v]\ \langle\Psi\rangle\{\Phi\}\}}{ewp_{\mathcal{E}}\ K[e]\ \langle\Psi\rangle\{\Phi\}}$$

$$\textsc{Try-With-Shallow}\quad \frac{ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} \qquad \mathit{shallow\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\}}{ewp_{\mathcal{E}}\ (\mathsf{shallow\text{-}try}\ e\ \mathsf{with}\ h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}$$

$$\textsc{Try-With-Deep}\quad \frac{ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} \qquad \mathit{deep\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\}}{ewp_{\mathcal{E}}\ (\mathsf{deep\text{-}try}\ e\ \mathsf{with}\ h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}$$

Fig. 6. Selected reasoning rules

$$\begin{array}{l}
\mathit{shallow\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad (\forall v.\ \Phi(v) \mathrel{-\!\!*} \triangleright\, ewp_{\mathcal{E}}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \land \\
\quad \forall v, k. \begin{cases} \Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{w.\ ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}\} \mathrel{-\!\!*} \\ \triangleright\, ewp_{\mathcal{E}}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathit{deep\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad (\forall v.\ \Phi(v) \mathrel{-\!\!*} \triangleright\, ewp_{\mathcal{E}}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \land \\
\quad \forall v, k. \begin{cases} \Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{w.\ \forall \Psi'', \Phi''. \\ \quad \triangleright\, \mathit{deep\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi''\rangle\{\Phi''\} \mathrel{-\!\!*} \\ \quad ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi''\rangle\{\Phi''\} \\ \} \mathrel{-\!\!*} \\ \triangleright\, ewp_{\mathcal{E}}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\} \end{cases}
\end{array}$$

Fig. 7. Definitions of the predicates *shallow-handler* and *deep-handler*

The law EWP3 describes the case where e is neither a value nor an effect. Then, we expect e to be able to make one step of computation. (Indeed, e would otherwise be "stuck". That would represent a runtime error, which we want to forbid.) Regardless of which step of computation is performed, we expect it to result in an expression e' that satisfies the specification $ewp_{\mathcal{E}}\ e'\ \langle\Psi\rangle\{\Phi\}$. More accurately, the operational semantics involves not expressions, but *configurations*, that is, pairs of an expression and a store. Following Jung et al. [2018], we let a fixed predicate S, known as the *state interpretation* predicate, encode an invariant about the store; for this reason, the right-hand side of the law involves an assumption $S(\sigma)$ and a goal $S(\sigma')$. Except for the presence of the protocol Ψ, this law is identical to Jung et al.'s [2018, §7], so we do not describe it further.

4.2 Reasoning Rules

We now establish a set of reasoning rules, which can be used to prove *ewp* assertions, and thereby prove properties of programs or program fragments. Each rule is a lemma that is stated and verified independently. A selection of rules appears in Figure 6. Each inference rule should be understood as a (universally quantified) magic wand.

4.2.1 Value, Do, Bind. Rule Value is a paraphrase of law EWP1 in Figure 5. It expresses the standard idea that one can return a value v if this value satisfies the current postcondition Φ.

Rule Do is obtained by specializing law EWP2 to the case where N is the empty evaluation context and by using law EWP1 to simplify the resulting assertion. It expresses the idea that one can perform an effect "do v" if the current protocol Ψ allows this request and guarantees that the reply satisfies the current postcondition Φ. A rule for reasoning about "do e" can be derived from Do and Bind. We believe that Do encourages reasoning about performing an effect essentially in the same way as one reasons about a calling a function, that is, in terms of a precondition and postcondition, *not* in terms of jumps and continuations.

Rule Bind allows reasoning about sequential composition. This rule reflects the fact that, in an expression of the form $N[e]$, the subexpression e is evaluated first. If it terminates normally by producing a value v, then this value is placed in the evaluation context N, yielding a new expression $N[v]$, which is then evaluated. The nesting of *ewp* assertions in the premise of Bind expresses the idea that the value v produced by e must be such that the expression $N[v]$ admits the postcondition Φ. This aspect of the Bind rule is standard. A novel aspect of this rule is its treatment of the protocol. If $N[e]$ must obey the protocol Ψ, then, according to the premise of the rule, both e and $N[w]$ must obey Ψ as well. This is again a consequence of our convention that every protocol is a repetition of a protocol step (§3.1.2): informally, in this rule, a repetition $\Psi^\star$ is split into a concatenation of repetitions $\Psi^\star; \Psi^\star$.

In the Bind rule, the evaluation context N is neutral: it cannot contain an effect-handling frame "shallow-try [] with $h \mid r$". To reason about such a frame, one can apply Try-With-Shallow or Try-With-Deep; or, if the expression e has no effect, one can apply Bind-Pure.

Rule Bind-Pure resembles Bind, but imposes an incomparable restriction. Whereas Bind can be applied to an arbitrary expression e but only to a neutral evaluation context N, Bind-Pure accepts an arbitrary evaluation context K but requires e to perform no effect. This requirement is expressed by an occurrence of the empty protocol $\perp$ in the premise. In a future extension of our system with multiple named effects, one can imagine unifying Bind and Bind-Pure into a single rule, with the condition that none of the effects performed by the expression e are handled by the context K.

4.2.2 Shallow Try/With. The rule Try-With-Shallow states that, if the expression e conforms to protocol Ψ and postcondition Φ, then, by wrapping it inside an effect handler whose arms are h and r, one makes it conform to a different protocol Ψ' and postcondition Φ'. The complexity of this rule is delegated to the auxiliary judgement $\textit{shallow-handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\}$. Intuitively, this judgement means that the handler $h \mid r$ acts as a "puzzle piece" whose "inner shape" is $\langle\Psi\rangle\{\Phi\}$ and whose "outer shape" is $\langle\Psi'\rangle\{\Phi'\}$. Its definition appears in Figure 7.

This judgement represents what one must prove when one wishes to establish that $h \mid r$ is a valid effect handler. It is an ordinary conjunction of two assertions.

The first conjunct, $\forall v.\ \Phi(v) \mathrel{-\!\!*} \triangleright\, \textit{ewp}_{\mathcal{E}}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}$, is a requirement on the second arm, r, which takes control when the expression monitored by the handler terminates normally and returns a value v. In that case, by EWP1, one may assume that $\Phi(v)$ holds, and one must prove that the function application $r\ v$ satisfies the outer shape $\langle\Psi'\rangle\{\Phi'\}$.

The second conjunct, $\forall v, k. \ldots$, is a requirement on the first arm, h, which takes control when the monitored expression performs an effect. Then, one must prove that the function application $h\ v\ k$ satisfies the outer shape $\langle\Psi'\rangle\{\Phi'\}$. In so doing, one cannot control v or k, which are chosen by the monitored expression, but one can assume that the request v is permitted by the protocol and that the continuation k can be safely applied to any reply w that is permitted by the protocol. Both assumptions at once are expressed by the assertion Ψ *allows* do v $\{w.\ \textit{ewp}_{\mathcal{E}}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}\}$.

Let us formulate a couple remarks about this assumption. First, the continuation application $k\ w$ is assumed to obey the protocol Ψ. Thus, after one effect has taken place, the protocol Ψ remains in force. This is again a consequence of our convention that every protocol is a repetition of a protocol step (§3.1.2): informally, a repetition $\Psi^\star$ is decomposed into $\Psi; \Psi^\star$. Second, the absence of a persistence modality $\square$ in front of $ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}$ means that this assumption about k can be exploited at most once. Therefore, the continuation can be invoked at most once. This reasoning rule enforces an affine usage of continuations.

There remains to explain why the definition of *shallow-handler* involves an ordinary conjunction, as opposed to a separating conjunction. Requiring a separating conjunction would be sound, but too strong. Indeed, a separating conjunction of magic wands $(P \mathrel{-\!\!*} P') * (Q \mathrel{-\!\!*} Q')$ allows both implications to be independently exploited, whereas an ordinary conjunction $(P \mathrel{-\!\!*} P') \wedge (Q \mathrel{-\!\!*} Q')$ is weaker and encodes an external choice: any one of the magic wands can be exploited, but not both. Here, an ordinary conjunction suffices because either h or r is invoked, depending on whether the monitored expression performs an effect or terminates normally, but not both. From the perspective of someone who must prove that a handler is correct, establishing an ordinary conjunction is easier. The same resources can be exploited in the proof that the first arm h is correct and in the proof that the second arm r is correct.

4.2.3 Deep Try/With. The construct "deep-try e with $h \mid r$" has been defined as syntactic sugar for an application of a recursive function, *deep*, which installs and repeatedly reinstalls a shallow handler (§2.2). The rule Try-With-Deep allows reasoning at a high level about this construct. It does not reveal that "deep-try" is defined in terms of lower-level constructs; the same rule would be used if "deep-try" was a primitive construct.

Try-With-Deep resembles Try-With-Shallow. The difference resides in the use of the auxiliary judgement $\textit{deep-handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\}$, whose definition appears in Figure 7. This judgement itself is defined very much in the same way as $\textit{shallow-handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi'\rangle\{\Phi'\}$. The only difference between these judgements resides in the assumption that one makes about the continuation k while proving the correctness of the first arm h of the handler. In the case of a shallow handler, this assumption involves the subformula

$$ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}$$

where w is universally quantified via Ψ *allows* do $v\ \{w. \ldots\}$, as the handler is free to choose any reply that is permitted by the protocol. In the case of a deep handler, this subformula is replaced with a slightly more complex one:

$$\forall \Psi'', \Phi''.\ \triangleright\, \textit{deep-handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi''\rangle\{\Phi''\} \mathrel{-\!\!*} ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi''\rangle\{\Phi''\}$$

This can be intuitively understood as follows. In the case of a shallow handler, the continuation invocation $k\ w$ obeys the "shape" $\langle\Psi\rangle\{\Phi\}$. In the case of a deep handler, the effect handler $h \mid r$ is recursively reinstalled inside the continuation, at the outermost level, thus changing the apparent shape of the continuation. Therefore, if one can prove that the effect handler $h \mid r$ adapts the inner shape $\langle\Psi\rangle\{\Phi\}$ to the outer shape $\langle\Psi''\rangle\{\Phi''\}$, then one can assume that the continuation invocation $k\ w$ obeys the latter shape.

Although this rule is quite technical, we believe that it is relatively easy to see, at such an abstract level, why it is correct. Let us emphasize a couple key aspects of it. First, the predicate *deep-handler* is recursively defined. This is a guarded recursive definition in the style of Iris [Jung et al. 2018, §5.6]. For this reason, when one wishes to establish a *deep-handler* assertion, one typically uses Löb induction. Lemma 5.1 illustrates this. Second, in the subformula that is displayed above, Ψ'' and Φ'' are universally quantified. They need not coincide with Ψ' and Φ'. This means that the outer shape exhibited by the effect handler $h \mid r$ is not fixed: every time this handler is recursively reinstalled, it may exhibit a new outer shape. This flexibility is required in some applications, such as the proof of `invert`, where the outer shape depends on a parameter *vs* that changes after every "yield" effect. This is visible in the statement of Lemma 5.1, which is universally quantified in *vs*.

4.2.4 Other Reasoning Rules. The rule Monotonicity states that the judgement $ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\}$ is covariant in both Ψ and Φ. It is analogous to Jung et al.'s wp-Wand [2018, §6.2]. As noted by these authors, it implies both the traditional consequence rule and the frame rule of Separation Logic. A more common statement of the frame rule is $P \ast ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} \vdash ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{P \ast \Phi\}$. As noted early on (§1), the frame rule does not come for free: it holds because our definition of *ewp* imposes an affine usage of continuations. More precisely, this stems from the absence of a persistence modality in law A4 (§3.3). In the case where the expression e is an effect "do v", the frame rule offers a strong guarantee: any resource P that is at hand before performing an effect (and that is not consumed by the very act of performing this effect) remains available afterwards. In addition to these aspects, Monotonicity allows exploiting the subsumption relation on protocols, $\Psi_1 \sqsubseteq \Psi_2$. In short, an expression that performs fewer effects can be placed where an expression that performs more effects is allowed.

The reasoning rules associated with standard programming language constructs (functions; products; sums; references) are not shown; they are standard. To give just one example, the rule for writing a reference is $\ell \mapsto v \ast \triangleright(\ell \mapsto w \mathrel{-\!\!\ast} \Phi(())) \vdash ewp_{\mathcal{E}}\ (\ell := w)\ \langle\Psi\rangle\{\Phi\}$.

Because our logic is an instance of Iris, all of the power of Iris is available to it; in particular, ghost state and invariants [Jung et al. 2018] can be used, if needed, exactly as usual.

4.3 Soundness

It is not difficult, based on our definition of *ewp* (Figure 5), to prove that our program logic is sound:

Theorem 4.1. *Let e be closed. If $\vdash ewp\ e\ \langle\bot\rangle\{\Phi\}$ holds then executing e in an empty heap is safe.*

If e can be verified under the rules of our program logic, with the full mask, with the empty protocol $\bot$ and with an arbitrary postcondition Φ, then executing e cannot lead to a stuck state. In other words, the execution of e must either diverge or terminate with a value; it cannot crash or terminate with an unhandled effect. Because our operational semantics includes a one-shot check, the absence of crashes implies that, during the execution of e, no continuation is invoked twice.

5 CASE STUDY: TURNING FOLDS INTO CASCADES

5.1 Folds and Cascades

One recurring fundamental question in software engineering is: how to perform iteration (that is, how to let a sequence of elements be transmitted from a producer to a consumer) while maintaining a modular separation between producers and consumers? There are many approaches to this problem: each proposes a different API to which producers and consumers must adhere. Two of the most prominent approaches, of interest in this section, are *folds* and *cascades*.

A *fold* is a producer represented as a higher-order function of type `('a -> unit) -> unit`, where `'a` is the type of the elements. In OCaml, for instance, the partial application of the standard library function `List.iter` to a list of integers is a fold: its type is `(int -> unit) -> unit`. In this representation, the producer is in control, and "pushes" elements towards the consumer.

A *cascade* is a producer represented as a lazy-list-like object that allows elements to be produced on demand. In the OCaml world, such an object is known under the name "sequence". The standard library module Seq makes the following type definition:

```
type 'a t = unit -> 'a head
 and 'a head = Nil | Cons of 'a * 'a t
```

```
1 let invert iter =
2   let yield = fun x -> do x in
3   fun () ->
4     deep-try iter yield with
5       fun x k -> Cons (x, k)  (* this arm handles an effect *)
6     | fun _   -> Nil          (* this arm handles normal termination *)
```

Fig. 8. Control inversion in *HH*

Thus, a cascade of type `'a Seq.t` is a function, a delayed computation. When applied to the value `()`, it returns either `Nil`, which means that the cascade is exhausted, or `Cons (x, xs)`, which means that `x` is the first element of the cascade and `xs`, another cascade, is its tail. In this representation, the consumer is in control, and "pulls" elements from the producer.

One may wonder: between folds and cascades, which representation is preferable? There is unfortunately no clear-cut answer. Instead, there is a tension: representing producers as folds usually makes them easier to implement, whereas representing them as cascades makes them easier to use and expands the range of their application scenarios.

5.2 From a Fold to a Cascade: Control Inversion

Fortunately, effect handlers (as well as other delimited control operators) provide a way of resolving this tension. By allowing a computation to be suspended and resumed on demand, they offer a simple way of transforming a fold into a cascade. This transformation can be implemented once and for all as a third-order function `invert`, whose code, expressed in the calculus *HH*, appears in Figure 8. A Multicore OCaml version of it exists online [de Vilhena and Pottier 2020a].

The code is extremely simple, or at least, seems so. The function `yield` defined on line 2 has type `a -> unit`, where `a` is the type of elements. All it does is "yield" an element `x` by performing an effect **`do`** `x`. The remaining lines (3–6) define the cascade returned by the function `invert`. The λ-abstraction **`fun`** `() -> ...` delays all action until the first element of this cascade is demanded by a consumer. Once this happens, the function call `iter yield` (line 4) is executed under a deep effect handler (lines 5 and 6). This function call is expected to produce a sequence of elements by applying `yield` to each element `x` in turn. Every time this happens, the computation is interrupted by an effect, the first arm of the handler takes control (line 5), and the value `Cons (x, k)`[4] is returned to the consumer. Thus, the consumer gets access to the element `x` and to a suspended computation, which in the consumer's eyes is just another cascade. If and when the consumer demands the next element of the cascade, by applying `k` to `()`, then the suspended computation is resumed, so the function call `iter yield` is allowed to make progress, until it either yields another element or terminates normally. In the latter case, the second arm of the handler takes control (line 6) and the value `Nil` is returned, indicating to the consumer that iteration is over.

5.3 Specification of Invert

What does `invert` do? The answer is simple: `invert` turns a fold into a cascade. Thus, if we have specifications in Separation Logic of folds and cascades, then we have a specification of `invert`.

As noted earlier (§3.2.4), following Filliâtre and Pereira [2016] and Pottier [2017], we parameterize the specifications of both folds and cascades with two predicates *permitted* and *complete* whose argument is a list of elements. Furthermore, we parameterize the specifications with an assertion C.

[4]We take the liberty of writing `Nil` for a left injection and `Cons` for a right injection.

$$\begin{array}{l} \mathit{isFold}\ \mathtt{iter} \triangleq \\ \quad \forall I, \Psi, f. \\ \qquad \Box\, (\forall us, u.\ \mathit{permitted}\,(us \mathbin{+\!\!+} [u]) \mathrel{-\!\!*} I\ us \mathrel{-\!\!*} \mathit{ewp}\ (f\ u)\ \langle\Psi\rangle\{().\ I\ (us \mathbin{+\!\!+} [u])\}) \mathrel{-\!\!*} \\ \qquad C \mathrel{-\!\!*} I\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\mathtt{iter}\ f)\ \langle\Psi\rangle\{().\ \exists us.\ I\ us \ast \mathit{complete}\, us \ast C\} \end{array}$$

$$\begin{array}{ll} \mathit{isCascade}\ k\ us \triangleq & \mathit{isCascadeHead}\ h\ us \triangleq \mathit{match}\ h\ \mathit{with} \\ \quad \mathit{ewp}\ k()\ \langle\bot\rangle & \quad \mid \mathit{Nil} \Rightarrow \mathit{complete}\, us \ast C \\ \qquad \{h.\ \mathit{isCascadeHead}\ h\ us\} & \quad \mid \mathit{Cons}\ (u, k') \Rightarrow \\ & \qquad \mathit{permitted}\,(us \mathbin{+\!\!+} [u]) \ast \triangleright \mathit{isCascade}\ k'\ (us \mathbin{+\!\!+} [u]) \\ & \quad \mid \_ \Rightarrow \mathsf{False} \end{array}$$

$$\vdash \mathit{isFold}\ \mathtt{iter} \mathrel{-\!\!*} C \mathrel{-\!\!*} \mathit{ewp}\ (\mathtt{invert}\ \mathtt{iter})\ \langle\bot\rangle\{k.\ \mathit{isCascade}\ k\ []\}$$

Fig. 9. Specification of folds, cascades, and `invert`

This assertion represents the access rights that the producer requires: it is typically an ownership assertion for a collection. Our specification and proof of `invert` are polymorphic in *permitted*, *complete*, and C.

Our specification of a fold appears in Figure 9. A fold is a second-order function `iter`, which takes a function f as an argument. The specification states roughly that if f processes one element, then `iter` f processes all elements. As in Pottier's paper [2017], it is polymorphic in a user-provided loop invariant I, which is parameterized by the elements produced so far, or "past elements". Initially, the user must prove $I\ []$, that is, prove that the invariant holds of the empty list. The user-provided function f must be able to move from $I\ us$ to $I\ (us \mathbin{+\!\!+} [u])$. At the end, the user obtains $I\ us$, where us is the list of all elements that have been produced. What is new here is that we parameterize the specification with a protocol Ψ, which appears in the specifications of both f and `iter` f. This conveys the idea that `iter` *itself does not perform any effects, nor does it handle any of the effects performed by* f. This information is crucial in the proof of `invert`.

Our specification of a cascade appears next in Figure 9. The mutually recursive predicates *isCascade* and *isCascadeHead* are parameterized with the past elements us, that is, with the elements already produced by this cascade. A cascade k is a function which can be applied (at most once) to a unit value. This application performs no effect and produces a cascade head h. A cascade head h is either $\mathit{Cons}\ (u, k')$, in which case the user can rely on the facts that (1) producing u at this point was permitted and (2) k' is a cascade that can produce the remaining elements, or *Nil*, in which case the user obtains the information that iteration is complete and regains access to the underlying collection, represented by the assertion C. The third branch $\mid \_ \Rightarrow \mathsf{False}$ is required because our calculus is untyped. Thus, we must explicitly exclude the possibility that a cascade head might be a value other than *Nil* or $\mathit{Cons}\ (\_, \_)$.

Because the definition of *isCascade* does not involve a persistence modality $\Box$, this predicate is affine: that is, a cascade can be used at most once.

The specification of `invert`, on the last line of Figure 9, states that `invert` performs no effect and transforms a fold into a cascade. The ownership of the collection, represented by C, must be abandoned when the cascade is created. Indeed, it is required by `iter`, which is invoked when the first element of the cascade is demanded. It is recovered once the cascade is exhausted.[5] The fact

[5] A perspicuous reader may notice that if the user stops iterating early and therefore does *not* exhaust the cascade, then she has no way of recovering the ownership of the collection. We believe that this problem could be remedied by imposing a linear usage of continuations, combined with a `discontinue` operation; see §8.

that a cascade is affine plays an important role in this specification: because we forbid a consumer from using a cascade twice, we are able to verify that none of the continuations that are captured inside the implementation of `invert` is invoked twice.

5.4 Verification of Invert

The verification of `invert` boils down to proving that the **`deep-try`** expression on line 4 conforms to the shape $\langle\bot\rangle\{h.\ \mathit{isCascadeHead}\ h\ []\}$. To prove this, we must apply Try-With-Deep (Figure 6). The main challenge is to find out with what loop invariant I and with what protocol the polymorphic specification of `iter` must be instantiated, so that (A) the function call `iter yield` on line 4 is safe and (B) the code in lines 5–6 forms a correct effect handler.

Let us momentarily postpone the choice of a loop invariant I, and consider the choice of a protocol. It should come as no surprise that a suitable protocol is $\mathit{SEQ}\ I$, where SEQ has been defined and explained earlier (§3.2.4), and where I is our yet-to-be-defined loop invariant. By choosing this protocol, we can immediately prove that the local function `yield` satisfies the specification

$$\Box(\forall xs,\ x.\ \mathit{permitted}\,(xs +\!\!+ [x]) \mathrel{-\!\!*} I\ xs \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{yield}\ ())\ \langle \mathit{SEQ}\ I\rangle\{I\ (xs +\!\!+ [x])\})$$

We immediately deduce that the application `iter yield` is safe and satisfies the specification

$$C \mathrel{-\!\!*} I\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{iter yield})\ \langle \mathit{SEQ}\ I\rangle\{().\ \exists us.\ I\ us \ast \mathit{complete}\, us \ast C\}$$

This takes care of point (A) above. There remains to choose a loop invariant I and address point (B).

The role of the invariant I is to keep track of the list of elements that have been yielded in the past. This is a purely logical role: the assertion $I\ us$ does not represent the ownership of a data structure in memory. Therefore, it seems natural to define I as ghost state. We allocate a ghost memory cell γ [Jung et al. 2018, §2.1] that holds a list us. More precisely, we use a combination of Iris's exclusive and authoritative monoids [Jung et al. 2018, §3.1,§6.3.3] so as to give rise to two ghost assertions $\boxed{\circ\ us}^{\gamma}$ and $\boxed{\bullet\ vs}^{\gamma}$ with the property that the conjunction $\boxed{\circ\ us}^{\gamma} \ast \boxed{\bullet\ vs}^{\gamma}$ implies $us = vs$ and can be updated (via a ghost update) to $\boxed{\circ\ us'}^{\gamma} \ast \boxed{\bullet\ us'}^{\gamma}$ where us' is arbitrarily chosen. As the loop invariant, we use $I\ us \triangleq \boxed{\circ\ us}^{\gamma}$. This assertion serves as Player's view of the current state.

Another assertion, $\boxed{\bullet\ vs}^{\gamma}$, serves as Opponent's view of the current state. It is apparent in the statement of Lemma 5.1, coming up. In the proof of this lemma, when Player and Opponent communicate via an effect, their views are confronted, the equality $us = vs$ follows, and both views are then updated to $us +\!\!+ [u]$.

Lemma 5.1. *The effect handler on lines 5–6 of Figure 8 is correct in the following sense:*

$$\begin{aligned}\forall vs.\ \boxed{\bullet\ vs}^{\gamma} \mathrel{-\!\!*} \mathit{deep\text{-}handler}\ \langle \mathit{SEQ}\ I\rangle\ &\{().\ \exists us.\ \boxed{\circ\ us}^{\gamma} \ast \mathit{complete}\, us \ast C\}\\ &(\textbf{fun}\ \texttt{x k -> Cons (x, k)}) \mid (\textbf{fun}\ \texttt{\_ -> Nil})\\ &\langle\bot\rangle\ \{h.\ \mathit{isCascadeHead}\ h\ vs\}\end{aligned}$$

In this statement, the assertion $\boxed{\bullet\ vs}^{\gamma}$ serves as Opponent's view, that is, the handler's view. Because *deep-handler* is recursively defined, this lemma must be proved by Löb induction. The assertion $\boxed{\bullet\ vs}^{\gamma}$ plays the role of an inductive assumption in this inductive proof. Therefore, it can also be regarded as a loop invariant: whereas $I\ us$ is `invert`'s view of the loop invariant, $\boxed{\bullet\ vs}^{\gamma}$ is the handler's view of the loop invariant.

By initializing the ghost cell γ with the empty list, and by instantiating Lemma 5.1 with the empty list, we obtain an instance of the above *deep-handler* judgement where vs is replaced with the empty list. This addresses point (B), and allows us to deduce that the **`deep-try`** expression on line 4 conforms to the shape $\langle\bot\rangle\{h.\ \mathit{isCascadeHead}\ h\ []\}$. This concludes the proof of `invert`.

```
(* type 'a status = Done of 'a | Waiting of ('a -> unit) list *)
(* type 'a promise = 'a status ref *)
(* effect Async : (unit -> 'a) -> 'a promise *)
(* effect Await : 'a promise -> 'a *)
let async e = do (Async e)
let await p = do (Await p)
let run main =
  let q = Queue.create() in
  let next() = if not (Queue.is_empty q) then Queue.take q in
  let rec fulfill p e =
    deep-try e() with
      (fun request k -> (* this arm handles an effect *)
        match request with
          Async e' ->
            let p' = ref (Waiting []) in
            Queue.add (fun () -> k p') q;
            fulfill p' e'
        | Await p' ->
            match !p' with
              Done v -> k v
            | Waiting ks -> p' := Waiting (k :: ks); next())
    | (fun v ->           (* this arm handles normal termination *)
        let Waiting ks = !p in
        List.iter (fun k -> Queue.add (fun () -> k v) q) ks;
        p := Done v;
        next())
  in
  fulfill (ref (Waiting [])) main
```

Fig. 10. A cooperative concurrency library in *HH*

6 CASE STUDY: A COOPERATIVE CONCURRENCY LIBRARY

Effect handlers are advertised as a modular foundation for effectful programming because they separate the description of the operations available to effectful programs from the implementation of these operations by handlers. Furthermore, they provide a structured interface to programming with delimited continuations. Dolan et al. [2017] illustrate these arguments by presenting an asynchronous I/O library whose implementation relies on effect handlers. This sort of application is in fact the primary motivation for introducing effect handlers in Multicore OCaml.

In this section, we specify and verify a slightly simplified[6] version of Dolan et al.'s library. Our version of the library involves higher-order functions, dynamically-allocated mutable state, and first-class continuations. Therefore, even though it fits in one page, its verification is quite challenging. We believe that this constitutes another good test and illustration of our reasoning principles. We begin with a brief explanation of the code (§6.1), followed with the specification (§6.2) and proof (§6.3) of the library.

[6] We remove the `yield` operation, which is not difficult to deal with. We also remove the code that deals with OCaml exceptions. We leave the combination of exceptions and effect handlers to future work (§8).

6.1 Implementation of the Library

The purpose of this library is to allow multiple user threads, conventionally known as *fibers*, to coexist, while allowing at most one fiber at a time to run. Two operations, `async` and `await`, are provided to the user so as to allow orchestrating fibers. "`async` e" spawns a new fiber, which executes the function application $e()$, and immediately returns a fresh *promise* p, which serves as a handle for the (future) result of this function application. "`await` p" blocks until the promise p has been fulfilled and returns its value.

The code of this library, expressed in *HH*,[7] appears in Figure 10. A Multicore OCaml version of it exists online [de Vilhena and Pottier 2020b]. The abstract notion of a promise, as well as the functions `async`, `await`, and `run`, are meant to be publicly visible; the rest is internal. The operations `async` and `await` are implemented by performing an effect (lines 5 and 6). The function `run` (line 7) is the scheduler: it runs the main fiber, represented by the function `main`, under a handler for the effects `Async` and `Await`. Therefore, whenever the active fiber calls `async` or `await`, this fiber is suspended and control is transferred to the scheduler, which manages promises and decides which of the suspended fibers should be resumed next.

The scheduler manages a number of promises, whose addresses are known to fibers. A promise is represented as a reference to a sum (line 2). Indeed, a promise either has or has not been fulfilled. In the former case, a value is stored. In the latter case, a list of fibers waiting on this promise is stored. Each such fiber is represented as a continuation.

The scheduler also maintains a FIFO queue `q` of ready fibers, that is, fibers that are currently suspended and are *not* waiting on any promise. This queue is initialized on line 8. Each fiber in it is represented as a function of type `unit -> unit`.

The main loop of the scheduler is implemented by the functions `next` and `fulfill`. The function call `next()` extracts an arbitrary ready fiber out of the ready queue and runs it (line 9). The purpose of `fulfill p e` is to execute the function call `e()`, while handling its `Async` and `Await` effects, and once it produces a value `v`, to store this value in the promise `p`, which thus becomes fulfilled. This is done by executing `e()` on line 11 under a handler whose arms handle three possible events:

(1) If `e()` performs an effect `Async e'`, then a new promise `p'` is created (line 15); the action of passing this promise to the continuation `k` is considered a new ready fiber (line 16), but is not scheduled immediately; instead, the newly spawned fiber becomes active (line 17). This is an arbitrary choice; one could do the converse.
(2) If `e()` performs an effect `Await p'`, then the status of the promise `p'` is examined. If it is fulfilled already, then its value `v` is immediately returned to the continuation `k` (line 20). Otherwise, `k` is added to the set of fibers waiting on `p'`, and an arbitrary ready fiber is scheduled (line 21).
(3) If `e()` terminates with a value `v` (line 22), the promise becomes fulfilled (line 25); the fibers `ks` that are waiting on this promise are retrieved (line 23) and become ready (line 24); and an arbitrary ready fiber is scheduled (line 26).

This process begins with an application of `fulfill` to a dummy promise and to the `main` fiber. It terminates when there are no more ready fibers (line 9), which means either that all fibers have finished or that there is a deadlock (a cycle of fibers that are waiting for one another).

The function `e` that is passed as an argument to `async`, as well as the function `main`, may perform `Async` and `Await` effects, whereas the functions stored in the queue `q`, which represent ready fibers, do not perform any effects. Indeed, these functions are captured continuations whose topmost

[7] Although *HH* is untyped, we provide some type information, inside comments, in the first four lines. We take the liberty of using the data constructors `Done` and `Waiting`, `Async` and `Await`, `[]` and `::` as sugar for left and right injections. The code contains free references to the external functions `List.iter`, `Queue.create`, `Queue.is_empty`, `Queue.take`, `Queue.add`.

$$isPromise : Val \to (Val \to iProp) \to iProp \qquad \text{persistent}(isPromise\ p\ \phi) \qquad Coop : iEff$$

$$\textsc{Async}\ \frac{ewp\ e()\ \langle Coop\rangle\{v.\ \Box\,\phi(v)\}}{ewp\ (async\ e)\ \langle Coop\rangle\ \{p.\ isPromise\ p\ \phi\}} \qquad \textsc{Await}\ \frac{isPromise\ p\ \phi}{ewp\ (await\ p)\ \langle Coop\rangle\ \{v.\ \Box\,\phi(v)\}} \qquad \textsc{Run}\ \frac{ewp\ main()\ \langle Coop\rangle\{\_.\ \text{True}\}}{ewp\ (run\ main)\ \langle\bot\rangle\ \{\_.\ \text{True}\}}$$

Fig. 11. Specification of the cooperative concurrency library

frame is an effect handler. Although this information is not clearly apparent in the code, it is visible in the specification of the library, which we present next.

6.2 Specification of the Library

Before we can verify this library, we must propose a specification for it. Since the publicly visible components of the library are the abstract notion of a promise and the functions `async`, `await`, and `run`, we expect this specification to publish at least an abstract predicate *isPromise* as well as three *ewp* assertions that indicate under what circumstances it is permitted to use `async`, `await`, and `run`. We propose such a specification in Figure 11.

Before explaining this specification, let us recall what guarantee we obtain by verifying that the library satisfies this specification: if a program that uses the library can be verified based on this specification, then this client program is safe: it cannot crash or perform an unhandled effect. There is no termination guarantee, since fibers are allowed to diverge. Moreover, deadlocks in fibers are not ruled out.

The specification begins with the declaration of an abstract predicate $isPromise\ p\ \phi$ (Figure 11). This assertion is persistent: a promise, once created, remains valid forever and can be awaited several times, from a single fiber or from distinct fibers. The intended meaning of this assertion is that p is a valid promise and that whatever value v can be obtained by waiting on this promise satisfies the assertion $\Box\,\phi(v)$. We use $\Box\,\phi(v)$, as opposed to $\phi(v)$, because it is permitted to await a promise twice: therefore, the assertion that one acquires by awaiting a promise must be duplicable. Persistence implies duplicability: every assertion of the form $\Box\,\phi(v)$ is duplicable.

The specification also declares the existence of a protocol *Coop* (Figure 11). It is an abstract protocol: its definition is not revealed. It appears in the specifications of `async` and `await`, which obey this protocol, and in the specification of `run`, whose argument `main` adheres to this protocol, whereas `run main` itself obeys the protocol $\bot$. This tells the user that `async` and `await` perform certain effects that `run` handles. This also forbids using `async` or `await` outside the dynamic extent of a call to `run`: indeed, since *Coop* is abstract, except by calling `run`, there is no way for the user to handle the effects performed by `async` and `await`.

Async states that if the function call `e()` returns a value v that satisfies the assertion $\Box\,\phi(v)$, then `async e` returns a promise p described by $isPromise\ p\ \phi$. A few aspects of this rule are worth emphasizing. First, in the premise, the expression `e()` is allowed to follow the protocol *Coop*. This means that the newly-spawned fiber may call `async` and `await`. Second, this rule is an implication $ewp\ \ldots \mathrel{-\!\!*} ewp\ \ldots$. This guarantees that the function call `e()` is executed at most once, and allows arbitrary resources to be transferred from the parent fiber to the newly-spawned fiber.

Await states that if p satisfies $isPromise\ p\ \phi$ then it is permitted to await p, and this operation returns a value v that satisfies $\Box\,\phi(v)$, if it returns at all.

Run states that if `main()` is safe, and possibly performs effects as per the protocol *Coop*, then `run main` is safe, and performs no effects. The postcondition of `run main` is True, which means that,

$$
\begin{aligned}
Coop \triangleq\ & \ \ Async\,\#\,!\,e\,\phi\,(e)\,\{\,\triangleright\, ewp\ e()\ \langle Coop\rangle\{v.\ \Box\,\phi(v)\}\,\}.\,?\,p\,(p)\,\{isPromise\ p\ \phi\} \\
& + Await\,\#\,!\,p\,\phi\,(p)\,\{isPromise\ p\ \phi\}.\,?\,v\,(v)\,\{\Box\,\phi(v)\}
\end{aligned}
$$

$$
\begin{aligned}
Ready\ q\ \phi\ k \triangleq\ & \forall v.\ \Box\,\phi(v) -\!\!*\ \triangleright\, PromiseInv\ q -\!\!*\ \triangleright\, isQueue\ q\ (Ready\ q\ (\lambda w.\ w = ())) -\!\!* \\
& ewp\ (k\ v)\ \langle\bot\rangle\{\_.\mathsf{True}\} \\
PromiseInv\ q \triangleq\ & \exists M.\ \boxed{\bullet M}^{\delta}\ * \\
& \forall\{(p,\gamma) \mapsto \phi\} \in M.\ \exists b.\ \boxed{\bullet b}^{\gamma}\ * \\
& \quad if\ b\ then\ \exists ks.\ p \mapsto Waiting\ ks\ *\ \forall k \in ks.\ Ready\ q\ \phi\ k \\
& \quad else\ \exists v.\ p \mapsto Done\ v\ *\ \Box\,\phi(v) \\
isPromise\ p\ \phi \triangleq\ & \exists\gamma.\ \boxed{\circ\,\{(p,\gamma) \mapsto \phi\}}^{\delta}
\end{aligned}
$$

Fig. 12. Internal definitions for the verification of the concurrency library

from the termination of `run main`, one can deduce nothing: in particular, one cannot deduce that the main fiber has completed its execution. Indeed, as explained earlier, if all fibers are in a deadlock, then the scheduler terminates, even though there exist unfulfilled promises and suspended fibers. Thus, one cannot replace the two occurrences of True in Run with two occurrences of an arbitrary assertion P. The specification, thus modified, would not hold.

6.3 Verification of the Library

Our concurrency library, shown in Figure 10, depends on two external modules, namely `List` and `Queue`. In Multicore OCaml, these modules are part of the standard library. To verify our library, we write specifications for the `List` and `Queue` modules, and we assume that these specifications are satisfied. The specification of lists is straightforward; we omit it. The specification of queues can be summarized as follows. An abstract predicate $isQueue\ q\ I$ means that q is (the address of) a well-formed queue, each of whose elements satisfies the predicate I. Creating a fresh queue produces an (empty) queue q such that $\forall I.\ isQueue\ q\ I$ holds.[8] If q satisfies $isQueue\ q\ I$, then inserting a value v into the queue consumes $I(v)$; conversely, successfully extracting a value v out of the queue yields $I(v)$. This is the specification of a bag: whether the queue is FIFO, LIFO, or otherwise is irrelevant.

In order to give an overview of the proof of the library, let us present a number of internal definitions, including the definition of the protocol *Coop*, the definition of the predicate *isPromise*, and the definition of the invariants that govern the scheduler's mutable data structures, namely the promises and the ready queue. These definitions appear in Figure 12.

The definition of the protocol *Coop* need not be examined in detail: indeed, the preconditions and postconditions assigned to the effects *Async* and *Await* in this protocol are exactly those that appear in the public specifications of the functions `async` and `await` in Figure 11. For this reason, the proofs of Async and Await are immediate as well. Therefore, the bulk of the verification work consists in verifying `run`.

The definitions of *Ready* and *PromiseInv* are guarded mutually recursive definitions.

The assertion $Ready\ q\ \phi\ k$ describes a fiber k, which may be either a suspended fiber that is waiting on a promise, or a ready fiber. It states that it is safe to execute the function application $k\ v$ under the three assumptions $\Box\,\phi(v)$ and $PromiseInv\ q$ and $isQueue\ q\ (Ready\ q\ (\lambda w.\ w = ()))$. It also

[8]This assertion is not duplicable; the universal quantifier can be instantiated only once. This description of queue creation allows instantiating I with an assertion that depends on q, a feature that is required here, as we wish to instantiate I with $Ready\ q\ (\lambda w.\ w = ())$. A similar situation arises in Timany and Birkedal's work [2019, §5.1].

guarantees that this function call performs no effect. The first assumption, $\Box\,\phi(v)$, is a condition on the value v. (If k is waiting on a promise, then ϕ is the postcondition of this promise; if k is ready, then ϕ is $\lambda w.\ w = ()$.) The second assumption requires every promise to be well-formed. The third assumption requires the ready queue to be well-formed and populated with ready fibers, each of which expects to be applied to a unit value (). The last two assumptions require the unique ownership of these data structures, and allow read and write access to them.

The assertion *PromiseInv q* states that there exists a collection of promises in the heap, each of which either is fulfilled (and stores a value) or is unfulfilled (and stores a list of waiting fibers). More precisely, with every promise p, we wish to associate two immutable pieces of information: (1) the address γ of a ghost cell whose content, a Boolean flag b, indicates whether this promise is unfulfilled; (2) the predicate ϕ that was provided by the user when this promise was created. The manner in which this is expressed in Iris does not really matter here, so we are brief. In short, a ghost cell at address δ stores a global map M whose entries have the form $(p, \gamma) \mapsto \phi$. For each such entry, a ghost cell at address γ holds a Boolean flag b. Depending on the value of this flag, an additional requirement is expressed:

(1) if b is true, which means that p is unfulfilled, then the reference cell at address p must contain *Waiting ks*, where every fiber k in the list *ks* is waiting on a value that satisfies ϕ.
(2) otherwise, this cell must contain *Done v*, for some value v such that $\Box\,\phi(v)$ holds.

Finally, the definition of *isPromise p* ϕ (still in Figure 12) states that there exists an entry of the form $(p, \gamma) \mapsto \phi$ in the map M. This guarantees that p is indeed the address of a promise.

Once these definitions are given, the bulk of the proof consists in proving the following lemma:

Lemma 6.1. *The function* `fulfill` *(line 10 of Figure 10) admits the following specification:*

$$\forall e, p, \gamma, \phi.\ \begin{cases} \mathit{PromiseInv}\ q \mathrel{-\!\!*} \mathit{isQueue}\ q\ (\mathit{Ready}\ q\ (\lambda w.\ w = ())) \mathrel{-\!\!*} \\ \boxed{\circ\,\{(p,\gamma) \mapsto \phi\}}^{\delta} \mathrel{-\!\!*} \boxed{\circ\,\mathsf{true}}^{\gamma} \mathrel{-\!\!*} \\ \mathit{ewp}\ e\ ()\ \langle \mathit{Coop} \rangle \{v.\ \Box\,\phi(v)\} \mathrel{-\!\!*} \\ \mathit{ewp}\ (\mathit{fulfill}\ p\ e)\ \langle \bot \rangle \{\_.\mathsf{True}\} \end{cases}$$

In short, this lemma states that if the promise p is unfulfilled and if $e()$ produces a value v that satisfies ϕ then the function call *fulfill p e* is safe and performs no effect. Inside the proof of this lemma, we apply the reasoning rule Try-With-Deep (Figure 6) and find an obligation to establish the following *deep-handler* judgement:

$$\begin{array}{l} \mathit{deep\text{-}handler}\ \langle \mathit{Coop} \rangle\ \{v.\,\Box\,\phi(v)\} \\ \qquad\qquad \text{(lines 12–21 in Figure 10)} \mid \text{(lines 22–26 in Figure 10)} \\ \qquad\qquad \langle \bot \rangle\ \{\_.\mathsf{True}\} \end{array}$$

The proof of this judgement is carried out by Löb induction. It is straightforward. The assumption that p is unfulfilled, encoded by the assertion $\boxed{\circ\,\mathsf{true}}^{\gamma}$, is exploited to argue that the nonexhaustive case analysis at line 23 cannot fail.

7 RELATED WORK

A great deal of work has been devoted to describing the semantics of effects and effect handlers in a pure setting—that is, either in the setting of an informal mathematical meta-language or via an embedding in a pure host programming language, such as Idris or Coq. Following Plotkin and Power [2004], many authors focus on an "algebraic" approach to reasoning, where one reasons essentially in terms of equalities between effectful computations. For instance, Plotkin and Pretnar [2008] propose a logic to reason about equality of computations in a calculus that has effects but no handlers. They later introduce effect handlers as an internal way of giving meaning to effects [2013] and discuss a notion of correctness whereby a handler is correct if it satisfies an intended equational theory. Brady [2013, 2014] embeds a programming language equipped with effect handlers as a domain-specific language inside Idris. The power of Idris's dependent type system allows assigning precise types to effectful computations. In Brady's later paper [2014], the sets of permitted effects at the beginning and end of an expression are not necessarily the same, and the final set of permitted effects may depend on the expression's result. FreeSpec [Letan et al. 2018] and Interaction Trees [Xia et al. 2020] can be described as embeddings of effects and effect handlers in Coq, together with libraries of lemmas that allow proving equalities between effectful computations. Xia et al. [2020, §8.2] offer an excellent survey of this area. Our work differs in its motivation from all of the papers cited above in that we wish to reason about a programming language that has both (1) user-defined effects and effect handlers and (2) primitive effects such as state and (in future work) concurrency. In such a setting, we believe that Separation Logic offers a more natural form of reasoning than equational reasoning.

Several researchers have proposed extensions of Hoare Logic with support for undelimited control operators—usually call/cc and throw. These include Berger [2009], whose calculus does not have any kind of mutable state; and Crolard and Polonowski [2012], whose calculus has stack-allocated mutable variables, but no dynamic allocation.

Delbianco and Nanevski [2013] propose HTT_{cc}, a Separation Logic for a calculus with dynamically-allocated mutable state and an "algebraic" variant of call/cc and throw. It uses "large footprint" assertions, which means that it imposes reasoning about the entire heap, as opposed to reasoning about a heap fragment and exploiting a frame rule to argue that the rest of the heap is preserved. It does not have a frame rule. If a program happens to preserve the part of the heap that it does not know about, then this must be explicitly stated in its specification, by universally quantifying over a residual heap [Delbianco and Nanevski 2013, §3]. Not all programs enjoy this property, though.

Most closely related to us, Timany and Birkedal [2019] develop an Iris-based Separation Logic for a calculus equipped with dynamically-allocated mutable state, concurrency, and call/cc and throw. Whereas we present a unary program logic, which can prove the safety of one program, they develop a binary framework, which can be used to establish a contextual refinement assertion between two programs.

Timany and Birkedal point out that "non-local control flow breaks the bind rule". In standard Separation Logic, this rule guarantees that one can reason about an expression e without caring under what context K this expression is evaluated. Timany and Birkedal define a predicate *wp* that does not have a bind rule, but allows a certain style of low-level reasoning: the reasoning rules that describe call/cc and throw paraphrase the operational semantics. On top of this, they define a "context-local weakest-precondition" predicate *clwp*, which does enjoy a bind rule, but is restricted to expressions that have no observable control effects. In contrast, in our system, which is aimed at reasoning about *delimited* continuations, both Bind and Bind-Pure hold: the logic reflects the fact that the expressions $K[e]$ and let $x = e$ in $K[x]$ are equivalent provided that none of the effects performed by e are handled by K. We believe that the key reason why we are able to separate reasoning about an effectful expression and reasoning about its context is that *we build a protocol, an explicit description of the dialogue between the expression and its context, into the predicate ewp.* This is new. As far as we are aware, no previous work extends Separation Logic with support for delimited control operators.

Actris [Hinrichsen et al. 2020] extends Iris with support for reasoning about concurrent programs that involve both shared mutable state and message-passing. Its "dependent separation protocols" allow a precise description of the dialogue and of the ownership transfers that take place across a communication channel. An early version of our work was very much inspired by Actris and used a similar form of protocols. Subsequently, though, we found that one can restrict one's attention to repetitive protocols (§3.1.2) without losing the ability to keep track of a protocol state. This is illustrated by several examples in §3.2 and in particular by the iteration protocol *SEQ I* that is used in the proof of `invert` (§3.2.4, §5.4). This restriction leads to a significant simplification in our judgements and reasoning rules.

Timany and Birkedal [2019] verify a cooperative concurrency library, which offers "fork" and "yield" operations, and is implemented using continuations. They prove that it refines a specification where "fork" and "yield" are primitive operations. We verify the safety of a library that offers a slightly richer set of operations, namely "async" and "await", and that is implemented using effect handlers. Kloos et al. [2015] propose a type system, inspired by Separation Logic, for a λ-calculus extended with primitive promises and `async` and `await` operations. Their promises are not duplicable. They prove the soundness of their type system with respect to an operational semantics, whereas we verify the correctness of an actual implementation of promises based on effect handlers. Leijen [2017] explores further applications of effect handlers in the area of asynchronous programming.

8 CONCLUSION AND FUTURE WORK

We have presented a powerful Separation Logic with support for user-defined effects and effect handlers, both shallow and deep. It is restricted to one-shot continuations. This logic enjoys two key rules that enable modular reasoning, namely the frame rule and a bind rule for neutral contexts. The ability to reason about an expression independently of the context in which it is placed stems from our use of a protocol that governs the dialogue between expression and context.

We have presented two small but nontrivial case studies, namely control inversion and an implementation of cooperative concurrency with promises. As far as we know, this is the first specification and proof of correctness of control inversion as a library. Our code and specification are particularly compact, generic, and tolerate mutable state. Our online repository [de Vilhena 2020] includes three more case studies, namely simulating exceptions as effects; implementing a single mutable memory cell as an effect using a parameterized handler; encoding shallow handlers into deep handlers.

Although we do not currently support reasoning about programs that involve primitive shared-memory concurrency, we do not envision a difficulty in extending our work with this feature. The reasoning rule for fork must impose the protocol $\perp$ on the newly-spawned thread: this guarantees that every effect performed by a thread is properly handled by this thread.

An important avenue for future work is to extend our system with support for multiple named effects. This requires introducing a mechanism for dynamically allocating new effect tags and allowing a handler to handle only a specific tag. Distinct handlers for distinct effects can then coexist. In such a setting, we do not yet know what exact form specifications and reasoning rules might take. In fact, even the choice of a suitable dynamic semantics is perhaps still a subject of debate [Biernacki et al. 2019; Zhang and Myers 2019; Brachthäuser et al. 2020].

Another important avenue for future work is to propose a system where one-shot and multi-shot continuations coexist. We envision restricting the frame rule to persistent assertions in areas of the code where multi-shot continuations are used. Finally, an intriguing idea is to view one-shot continuations as linear instead of affine. Every one-shot continuation would then have to be either invoked or explicitly discarded. In such a system, exceptions would not be regarded as a special case of effects. Instead, exceptions and effects would coexist, as in Multicore OCaml, and the primitive operation that discards a continuation would resume the suspended computation by raising an exception inside it. This would guarantee that every code block, once entered, must be exited, either normally or via an exception. This in turn would help programmers ensure that resources, once allocated, are properly deallocated.

REFERENCES

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Andrej Bauer and Matija Pretnar. 2020. Eff.

Martin Berger. 2009. Program Logics for Sequential Higher-Order Control. In *Fundamentals of Software Engineering (Lecture Notes in Computer Science)*, Vol. 5961. Springer, 194–211.

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.

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.

Edwin C. Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming (ICFP)*. 133–144.

Edwin C. Brady. 2014. Resource-Dependent Algebraic Effects. In *Trends in Functional Programming (TFP) (Lecture Notes in Computer Science)*, Vol. 8843. Springer, 18–33.

Tristan Crolard and Emmanuel Polonowski. 2012. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *Journal of Logical and Algebraic Methods in Programming* 81, 3 (2012), 181–208.

Paulo Emílio de Vilhena. 2020. A Separation Logic for Effect Handlers: Coq formalization. https://gitlab.inria.fr/pdevilhe/hazel.

Paulo Emílio de Vilhena and François Pottier. 2020a. Control inversion in Multicore OCaml. https://gitlab.inria.fr/pdevilhe/hazel/-/blob/master/src/invert.ml.

Paulo Emílio de Vilhena and François Pottier. 2020b. Cooperative concurrency in Multicore OCaml. https://gitlab.inria.fr/pdevilhe/hazel/-/blob/master/src/promises.ml.

Paulo Emílio de Vilhena and François Pottier. 2020c. Problems with multi-shot continuations in Multicore OCaml. https://gitlab.inria.fr/pdevilhe/hazel/-/blob/master/src/test.ml.

Germán Andrés Delbianco and Aleksandar Nanevski. 2013. Hoare-style reasoning with (algebraic) continuations. In *International Conference on Functional Programming (ICFP)*. 363–376.

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.

Stephan Dolan, Anil Madhavapeddy, and KC Sivaramakrishnan. 2020. Multicore OCaml.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* 22, 4-5 (2012), 477–528.

Jean-Christophe Filliâtre and Mário Pereira. 2016. A Modular Way to Reason About Iteration. In *NASA Formal Methods (NFM) (Lecture Notes in Computer Science)*, Vol. 9690. Springer, 322–336.

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 11275. Springer, 415–435.

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020), e5.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 6:1–6:30.

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.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *International Conference on Functional Programming (ICFP)*. 145–158.

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous Liquid Separation Types. In *European Conference on Object-Oriented Programming (ECOOP)*. 396–420.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Workshop on Mathematically Structured Functional Programming (MSFP)*, Vol. 153. 100–126.

Daan Leijen. 2017. Structured asynchrony with algebraic effects. In *Type-Driven Development (TyDe)*. 16–29.

Daan Leijen. 2020. Koka.

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *Formal Methods (FM) (Lecture Notes in Computer Science)*, Vol. 10951. Springer, 338–354.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Principles of Programming Languages (POPL)*.

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

Gordon D. Plotkin and A. John Power. 2004. Computational Effects and Operations: An Overview. *Electronic Notes in Theoretical Computer Science* 73 (2004), 149–163.

Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Logic in Computer Science (LICS)*. 118–129.

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.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013).

François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *Certified Programs and Proofs (CPP)*. 3–16.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74.

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.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 51:1–51:32.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 5:1–5:29.