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 sequence of 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, and so on. 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:

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 effect handler. An effect handler is very much like an exception handler, with one key difference: unlike an exception handler, an effect handler 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 offer the ability of capturing part of the evaluation context and reifying it as a continuation, effects and effect handlers are a form of delimited control.

Effect handlers are found in a number of 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]. Thus, there is growing interest for effect handlers, as well as a growing need for programmers and researchers...
to understand how to reason about them. Furthermore, several of these programming languages have primitive effects such as input/output, dynamically-allocated mutable state, and concurrency. There is therefore a need to understand how to reason about the combination of primitive effects and user-defined effects and 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 most programming teachers are aware that the 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 reason in terms of 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 impose the rule that a continuation must be invoked at most once. In Multicore OCaml [Dolan et al. 2017], for instance, this is enforced via a runtime check. One might think that this restriction is motivated by obvious performance considerations: supporting multiple invocations of a continuation is likely to involve copying stack segments, an expensive operation. In reality, though, there is a less obvious, more fundamental reason why one may wish to impose this restriction: allowing continuations to be invoked more than once breaks certain fundamental laws of reasoning about programs.

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. For instance, in the following code snippet, expressed in a hypothetical variant of Multicore OCaml that allows multi-shot continuations, the assertion on the last line is wrong:

```ocaml
let x = ref 0 in  (* initialize x to zero *)
f();  (* the unknown function f has no access to x *)
x := !x + 1;  (* increment x, apparently from zero to one *)
assert (!x = 1);  (* so x must contain one, right? no: wrong! *)
```

If the function f captures its continuation and if this continuation is invoked twice, then the mutable variable x can be incremented twice, first from 0 to 1, then from 1 to 2, causing the runtime assertion to fail. So, the informal reasoning in the comments is wrong. Yet, it is based on the most central rules of Separation Logic, namely the rule of sequential composition, also known as the “bind rule”, and the frame rule. Indeed, in the example above, these rules allow arguing that the assertion $x \mapsto 0$
is preserved through the call to \( f \) and deducing that the assertion on the last line is correct. In particular, the frame rule alone seems to clearly guarantee that “a code block, once entered, is exited at most once”. Thus, **multi-shot continuations break the frame rule**, a remark that we have never seen in print. Dreyer et al. [2012] do note that call/cc breaks the “well-bracketing” of computations, but their paper does not involve Separation Logic. Timany and Birkedal [2019] note that call/cc breaks the bind rule. The situation in our setting is different, because we are interested in delimited continuations. We come back to this point in §7.

In conclusion, multi-shot continuations have a performance impact and break a reasoning law that programmers are accustomed to relying upon. If they are allowed, then the frame rule must be removed or restricted. We choose to impose one-shot continuations and keep the frame rule.

**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.** Before explaining how we wish to think about effects (§2), let us summarize our contributions. We develop a Separation Logic, based on Iris, for a call-by-value \( \lambda \)-calculus equipped with primitive heap-allocated mutable state and with effect handlers (§3). By design, we forbid multi-shot continuations. Our reasoning rules (§4) include the key rules of Separation Logic, including the 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 inspired by Dolan et al. [2017] (§6). Our proofs are machine-checked [Anonymous 2020]. The current limitations of our work, discussed at the end of the paper (§8), include the absence of shared-memory concurrency and the absence of multiple named effects.

## 2 PROTOCOLS AND SPECIFICATIONS

Before inventing reasoning rules for effects and handlers, we must determine under what form we expect to write specifications. A specification should describe 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, and no further.

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 \( \Phi \), 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 terminates and 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 very much akin to the manner in which two processes might communicate over a channel while obeying a pre-agreed protocol. In the following, we introduce a syntax of protocols (§2.1), which is inspired by Hinrichsen et al.’s dependent session protocols [2020], and we give several examples of protocols (§2.2). Then, we come back to the manner in which protocols appear in the specification of an expression (§2.3).

2.1 Syntax of Protocols

In our setting, what form might a protocol take? 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 $v$ is a value, which 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. In that case, the execution of the expression is resumed, and the dialogue continues. The manner in which it may continue depends, in the most general case, on the request $v$ and reply $w$ that have been exchanged.

Because a protocol describes a sequence of requests and replies, one might let a protocol $\Psi$ be just a list of pairs of values. The syntax of protocols would then be:

$$\Psi ::= \text{end} | ! (v).? (w). \Psi \quad \text{(tentative syntax of protocols, take 1)}$$

The protocol “end” indicates that no effects are allowed. The request/reply protocol $! (v).? (w). \Psi$ means that the expression may make the request $v$ (it may also choose to make no request at all), that the environment may reply with $w$ (it may also choose to never reply), and that the rest of the dialogue is governed by the protocol $\Psi$.

This syntax is a good start, but is not yet expressive enough for our purposes. For one thing, it says nothing about the global state; yet, in a programming language equipped with primitive mutable state, it is necessary to be able to describe how the state evolves over time. More generally, in a Separation Logic, it is necessary to be able to describe transfers of ownership from Player to Opponent and back. To achieve both goals at once, we enrich the syntax of protocols with pre- and postconditions, as follows:

$$\Psi ::= \text{end} | ! (v) \{P\}.? (w) \{Q\}. \Psi \quad \text{(tentative syntax of protocols, take 2)}$$

The protocol $! (v) \{P\}.? (w) \{Q\}. \Psi$ means that, if and when the expression makes the request $v$, the assertion $P$ must hold, and the ownership of the resources governed by $P$ is transferred from Player to Opponent; symmetrically, if and when the environment replies with $w$, the assertion $Q$ must hold, and the ownership of the resources that it controls is transferred from Opponent to Player.

One last missing ingredient is the ability for Player to choose between several permitted requests and the dual ability for Opponent to choose between several permitted replies. We introduce this
flexibility by extending the syntax of protocols with binders $\bar{x}$ and $\bar{y}$:

$$\Psi ::= \text{end} \mid \! \bar{x}(v) \{P\}. \ ? \bar{y}(w) \{Q\}. \ \Psi$$

(definition of syntax of protocols)

In a request/reply protocol $\! \bar{x}(v) \{P\}. \ ? \bar{y}(w) \{Q\}. \ \Psi$, the scope of the binders $\bar{x}$ and $\bar{y}$ extends all the way towards the right. When making a request, Player may choose any instance of the variables $\bar{x}$, provided the assertion $P$ (which may depend on $\bar{x}$) is satisfied. Symmetrically, when replying, Opponent may choose any instance of the variables $\bar{y}$, provided $Q$ (which may depend on $\bar{y}$) is satisfied. Because $\Psi$ may depend on both $\bar{x}$ and $\bar{y}$, the remainder of the dialogue is influenced by these choices. This is therefore a dependent protocol [Hinrichsen et al. 2020].

In order to be able to describe a potentially infinite interaction, we interpret this definition in such a way that a protocol may be infinite. Furthermore, in some of the examples that follow, we use protocols of the form $\Psi + \Psi$. This is an external choice construct: in the protocol $\Psi_1 + \Psi_2$, it is up to Player to choose between $\Psi_1$ and $\Psi_2$. Choice can be defined as a metalevel construct, without extending the syntax of protocols. We give more details about these aspects later on ($\S$4.3).

### 2.2 Examples of Protocols

#### 2.2.1 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:

$$\Psi_{\text{abort}} \triangleq \! () \{\text{True}\}. ? y(y) \{\text{False}\}. \text{end}$$

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. Thus, the remainder of the protocol, end, is irrelevant.

#### 2.2.2 Memory Cell with Exchange.

Suppose that there exists a mutable memory cell at location $\ell$ in the heap, and suppose that, instead of offering direct access to this location, one wishes to install a handler for an “exchange” effect, which allows the program to write a new value into the memory cell and (at the same time) to obtain its previous value. Because this handler updates a memory cell in the heap, the protocol that describes this effect must involve nontrivial pre- and postconditions.

One approach is to let a concrete “points-to” assertion appear in the pre- and postcondition. This leads to the following protocol:

$$\Psi_{\text{xchg}} \triangleq \! x \ x'(x') \{\ell \mapsto x\}. \ ? (x) \{\ell \mapsto x'\}. \ \Psi_{\text{xchg}}$$

This protocol makes use of the ability to bind variables. Here, two variables $x$ and $x'$ are bound. They can be intuitively regarded as universally quantified: indeed, when performing an xchg effect, one can instantiate $x$ and $x'$ with two arbitrary values $v$ and $v'$. Thus, this protocol states that, provided the location $\ell$ currently contains some value $v$, the expression may perform the effect “do $v'$”. If the handler chooses to reply, then it must reply by returning the value $v$ and by updating the memory location $\ell$ with the value $v'$. During this interaction, the unique ownership of the memory location is transferred from the expression to the handler, and back. The continuation of the protocol $\Psi_{\text{xchg}}$ is itself: this protocol is recursive. This means that the xchg effect can be performed as many times as one wishes, possibly infinitely many times.
A slightly more elegant approach is to abstract the protocol over the assertion “ℓ ➔ _”, by replacing this assertion with a predicate I of type “value to assertion”:

\[ Ψ_{\text{xchg}} I ≜ ! \ x \ x' (x') \ {I \ x}. ? (x) \ {I \ x'}. \ Ψ_{\text{xchg}} I \]

An expression that makes use of the xchg effect can then be required to be polymorphic in I. This guarantees that it is unaware of the manner in which this effect is implemented, and that it cannot read or write the memory cell except via the xchg effect.

2.2.3 Memory Cell with Read and Write. The previous example uses a single xchg effect to perform a combination of a read and a write. It is possible to separate these operations, at the cost of a slightly more complex protocol. Let us write “read” for the left injection inj₁ () and “write v’” for the right injection inj₂ v’. A request must be either read, in which case the environment must reply with the current value 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, which we again parameterize over I:

\[ Ψ_{r/w} I ≜ ! \ x \ (\text{read}) \ {I \ x}. ? (x) \ {I \ x}. \ Ψ_{r/w} I \\
+ ! x \ x' (\text{write} \ x') \ {I \ x}. ? (() \ {I \ x'}. \ Ψ_{r/w} I \]

2.2.4 Sequence of Elements. Suppose one wishes to allow an expression to produce a sequence of elements. An element x is produced by performing an effect “do x”, and this can be repeated as many times as desired. If the sequence xs of elements that must be produced in the future is determined in advance, then the interaction between the producer and its environment is described by the protocol Ψ_seq xs, where the parameterized protocol Ψ_seq is recursively defined as follows:

\[ Ψ_{\text{seq}} [] ≜ \text{end} \]

\[ Ψ_{\text{seq}} (x :: xs) ≜ ! (x) \ {(\text{True}). ? (() \ {\text{True}). \ Ψ_{\text{seq}} xs} \]

This definition involves a meta-level case analysis. In the case of an empty sequence, no elements may be produced, so no effects may be performed: the protocol is “end”. In the case of a nonempty sequence x :: xs, the expression may perform the effect “do x”, to which the environment must reply with the value (); the remainder of the interaction is described by Ψ_seq xs.

2.3 Protocols in Specifications

In ordinary Separation Logic, a specification is a triple \{P\} e \{v, Q\}, where P and Q are assertions that describe the initial and final state, and the bound metavariable v lets the postcondition Q refer to the value returned by the expression e. In Iris [Jung et al. 2018, §6], such a triple is in fact defined as sugar for ⊩ (P ➔ wp e \{v, Q\}), where wp is the weakest precondition predicate, and where the persistence modality □ indicates that a triple must be persistent [Jung et al. 2018, §5.3]. Jung et al. argue quite convincingly that working with magic wands and wp assertions is often simpler and more powerful than working at the level of triples.

The use of the persistence modality 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 ➔ wp k() \{v, Q\} 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 way protocols should appear in specifications. We formulate two remarks:

1. Quite obviously, a specification should contain a protocol $\Psi$, which indicates what requests may be sent and what replies may be received by the Player. So, we might aim to define a weakest-precondition predicate along the lines of “$\text{ewp } e \langle \Psi \rangle \{v, Q\}$.”

2. When reasoning about a sequential composition $e_1; e_2$, we must allow a prefix of the protocol to be carried out by $e_1$, while the remainder of the protocol is carried out by $e_2$. To achieve this, we let the assertion $\text{ewp } e \langle \Psi \rangle \{\ldots\}$ mean that $e$ must abide by the protocol $\Psi$, but does not have to carry out all of it: it is permitted for $e$ to carry out only a prefix of $\Psi$, while the remainder $\Psi'$ must be carried out by the expression that follows $e$. Naturally, the specification of $e$ must offer a description of $\Psi'$, and (in general) must allow $\Psi'$ to depend on the return value and final state produced by the execution of $e$. We achieve this by letting the postcondition depend on $\Psi'$. We adopt a weakest-precondition assertion of the form $\text{ewp } e \langle \Psi \rangle \{(v, \Psi'). Q\}$, where $\Psi$ is the initial protocol and where the assertion $Q$ imposes a constraint at the same time on the result value $v$, on the protocol remainder $\Psi'$, and (implicitly) on the global state.

The assertion $\text{ewp } e \langle \Psi \rangle \{(v, \Psi'). Q\}$ represents a permission to perform the effects described by the protocol $\Psi$; there is a priori no obligation to follow one of the branches of the protocol until the end. Yet, it is possible to express such an obligation. By letting $\Psi' = \text{end}$ appear in the postcondition, for instance, one can require a certain sequence of interactions to be carried out until its end. In such a case, divergence remains permitted (because this is a logic of partial correctness) but normal termination is forbidden unless the protocol has been entirely carried out.

3 SYNTAX AND SEMANTICS OF $HH$

We present $HH$, a call-by-value $\lambda$-calculus equipped with primitive mutable state and effect handlers. Its name stands for ”heaps and handlers”. Its syntax and operational semantics are standard. A reader who is familiar with effect handlers may skim through this section.

3.1 Syntax

The syntax of $HH$ appears in Figure 1. Its features include primitive values and operations, recursive functions, binary products and sums, mutable references, and constructs for performing and handling effects. There is no “let” form: sequential composition is encoded as a $\beta$-redex.

The construct “$\text{do } v$” 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 $(\lambda N)$. The handler receives both the value $v$ and this continuation as arguments. Thus, the value $v$ is transmitted from the computation to the handler.

The construct do $v$ is sugar for $\$([],)[do v]$, where $[]$ is the empty context. The more general construct $\$([N])[do v]$ is a do construct that has already captured the (partial) evaluation context $N$. It plays a role in the small-step operational semantics and is not accessible to the programmer. In the literature, it is often written op $v N$ [Plotkin and Pretnar 2009; Kammar et al. 2013].

The construct “shallow-try $e$ with $h | r$” 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.

---

1We name our weakest-precondition predicate $\text{ewp}$ so as to better distinguish it from the standard $wp$. 

\[ n \in \mathbb{Z} \quad \ominus \in \{+,-,\ldots\} \quad v,k,r \mapsto \mu f.\lambda x. e | \ell | (v,v) | \text{inj}_i v | (\lambda N) \]
equation
defines integer values
primitive operations (of arity 2)
values
expressions
\[
\lambda\text{-calculus}
\]
primitive values & operations
pairs
sums and case distinction
memory locations
effects
effect handlers
evaluation contexts
evaluation under shallow-try
standard evaluation contexts
neutral evaluation contexts
standard evaluation contexts

\[
| 1 | (\lambda v_2)[do v_2] / \sigma \rightarrow \Lambda(e_1 \circ N)[do v_2] / \sigma
\]
shallow-try
\[
| (\Lambda v_1)[do v_1] \circ v_2 / \sigma \rightarrow \Lambda(N \circ v_2)[do v_1] / \sigma
\]
shallow-try
\[
| \Lambda(\lambda N) \circ v / \sigma \rightarrow \Lambda N[v] / \sigma
\]

3.2 Encoding Deep Handlers
The construct "deep-try e with \( h \mid r \)", which installs a deep handler, is sugar for the function application \( \text{deep}(\lambda(). e) h r \), where the recursive function \( \text{deep} \) is defined as follows:
\[
\text{deep} \triangleq \mu \text{deep}.\lambda e h r. \text{shallow-try } e() \text{ with } \lambda v k. h v (\lambda x. \text{deep}(\lambda(). k x) h r) \mid r
\]
Instead of passing to the handler the continuation \( k \), we pass it \( \lambda x. \text{deep}(\lambda(). k x) h r \), 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 \cite{Hillerström2018}*§3.1.

3.3 Connecting HH and Multicore OCaml
This paper presents a program logic for HH, and the programs that we verify (§5, §6) are HH programs. Yet, we prefer to present these programs in the syntax of Multicore OCaml 4.10.0 (Figures 7 and 9): this is more readable and allows this code to be type-checked and executed. In Multicore OCaml, perform performs an effect; "match e with effect ..." installs a deep handler; continue converts a continuation into a function. (In HH, no such conversion is required.)
3.4 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 \( \sigma \), which are finite maps of memory locations to values. Some of the rules that define the head reduction relation \( e / \sigma \rightarrow e' / \sigma' \) appear in Figure 2.

The first two rules in Figure 2 allow the construct \( \text{§}(N)[\text{do } e] \) to capture one frame of the evaluation context that surrounds it. There are more rules in this style (not shown). These rules allow a “do” construct 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.

The next two rules in Figure 2 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 \( (\lambda N) \). If a value \( v \) is returned, the second branch of the handler takes control and receives \( v \) as an argument.

The last rule in Figure 2 defines the behavior of a first-class continuation \( (\lambda N) \): when it is applied to a value \( v \), it reduces to \( N[v] \). Thus, if and when the handler decides to apply the captured continuation to a value \( v \), the suspended computation is resumed, just as if the expression “do \( v \)” had returned the value \( v \). Invoking a continuation more than once does not cause a runtime error. We return to this point later on (§4.4).

The “neutral” evaluation contexts \( N \) that appear in the reduction rules are defined in Figure 1. They do not include the form “shallow-try [] with \( h \mid r \)”. Indeed, in this calculus, the evaluation context is always captured up to the nearest enclosing handler; this implies that an active handler is never captured inside a continuation.

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 \( e / \sigma \rightarrow e' / \sigma' \) is permitted, then the reduction step \( K[e] / \sigma \rightarrow K[e'] / \sigma' \) is permitted. This allows reduction to take place under an active handler.

4 A SEPARATION LOGIC FOR HH

As explained earlier (§2), we wish to work with specifications of the form \( \text{ewp } e \langle \Psi \rangle \{ \Phi \} \), where \( e \) is an expression, \( \Psi \) describes the effects that \( e \) may perform, and \( \Phi \) describes the situation that results—that is, the return value, the final state, and the remainder of the interaction protocol. In the Iris tradition [Jung et al. 2018], we follow a semantic approach: first, we give direct meaning to the assertion \( \text{ewp } e \langle \Psi \rangle \{ \Phi \} \) (§4.1); then, we establish a number of lemmas, also known as reasoning rules, that can be used to prove \( \text{ewp} \) assertions (§4.2). We use the syntax of protocols that was presented earlier (§2.1); how this syntax is embedded in Coq is described later on (§4.3).

4.1 Specifications and Their Meaning

The definition\(^2\) of \( \text{ewp} \) appears in Figure 3. It is presented as a set of four defining laws. Law (1) covers the case where the expression \( e \) is a value. Laws (2) and (3) cover the case where \( e \) is an effect, that is, an expression of the form \( \text{§}(N)[\text{do } v'] \). Law (4) covers the remaining cases.

Law (1) defines what it means for a trivial computation, which immediately returns a value \( v \), to satisfy protocol \( \Psi \) and postcondition \( \Phi \). Even though such a computation performs no effect,

\(^2\)We have superficially simplified this definition by hiding all “later” modalities (which may be required, among other reasons, to justify that the definition is well-formed) and all “update” modalities (which allow ghost state updates to take place at suitable times). For details, the reader is referred to Jung et al. [2018] and to our formal proofs [Anonymous 2020].
this law does not require \( \Psi \) to be the trivial protocol, end. Indeed, in an assertion \( \text{ewp } e \langle \Psi \rangle \{ \Phi \} \), the protocol \( \Psi \) represents the effects that the expression \( e \) may perform. What law (1) does require is that the postcondition \( \Phi \) must hold. Whereas, in standard Separation Logic, a postcondition is a function of a value to an assertion, which means that it constrains the return value \( v \) and (implicitly) the global state, in this logic, it is a function of a value and a protocol to an assertion, which means that it constrains not only the return value and final state, but also the remainder of the interaction protocol. In law (1), because no effects are performed by the expression \( v \), the remainder of the protocol is all of it. Thus, the right-hand side of the law is \( \Phi (v, \Psi) \).

Laws (2) and (3) define the meaning of the assertion \( \text{ewp } e \langle \Psi \rangle \{ \Phi \} \) when \( e \) is an effect, that is, a construct of the form \( \text{§}(N)[\text{do } v'] \langle P \rangle \), where \( v' \) is the value that is passed from the expression to the handler, and \( N \) is the captured evaluation context. Law (2) is simple: because the protocol \( \text{end} \) disallows all effects, if \( \Psi \) is \( \text{end} \) then this assertion is false. Law (3) governs the more interesting case where the protocol is \( ! \bar{x} (v) \{ P \} \langle Q \rangle \). Such a protocol allows an effect to take place. It means intuitively that Player may instantiate the auxiliary variables \( \bar{x} \) however it pleases; it must then send the request \( v \) and relinquish the resources described by the assertion \( P \). (Because \( \bar{x} \) may appear in \( v \) in \( P \), Player may have a choice between multiple permitted requests.) Accordingly, the right-hand side of law (3) begins with \( \exists \bar{x}. v' = v * P * \ldots \), which means that the request \( v' \) must have the required shape (namely, an instance of \( \bar{x} \)) and that, at the time the request is made, the assertion \( P \) must hold. Furthermore, the protocol intuitively means that if Opponent decides to reply (by invoking the captured continuation), then it must do so by instantiating the variables \( \bar{y} \) however it pleases, by sending the reply \( w \), and by relinquishing the resources \( Q \). This explains the second part of the right-hand side of law (3), \( \forall \bar{y}. Q \rightarrow \text{ewp } N[w] \langle \psi \rangle \{ \Phi \} \); whatever \( \bar{y} \) is chosen by Opponent, Player (represented here by the evaluation context \( N \)) must be prepared to handle the reply \( w \). Furthermore, Player may assume that the assertion \( Q \) holds, and must behave as dictated by the remainder \( \psi \) of the protocol and by the postcondition \( \Phi \).

Law (3) does not involve the persistence modality \( \Box \) [Jung et al. 2018, §5.3]. This means that Player, who is in charge of establishing the implication \( Q \rightarrow \text{ewp } N[w] \langle \psi \rangle \{ \Phi \} \), can assume that this implication will be exploited at most once. Thus, a nonpersistent resource can be used in the proof of this implication. This means that such a resource can be transmitted across a “do” statement: in other words, the frame rule can be applied to a “do” statement! Dually, the absence of a persistence modality implies that Opponent can invoke a captured continuation at most once. This is where we articulate our decision to forbid multi-shot continuations and (in exchange for this restriction) preserve the frame rule (§1).

Law (4) describes the case where the expression \( e \) 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 $\text{ewp } e' \langle \Psi \} \{\Phi\}$. More accurately, the operational semantics involves not expressions, but \textit{configurations}, that is, pairs of an expression and a store. Following Jung et al. [2018], we let a fixed predicate $S$, known as the \textit{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 $\Psi$, 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 collection of reasoning rules, which can be used to prove $\text{ewp}$ assertions, and thereby to prove properties of programs or program fragments. Each reasoning rule is a lemma that is stated and verified independently. Some of these rules appear in Figure 4. Each inference rule should be understood as a (universally quantified) magic wand.

4.2.1 \textbf{Do}. Rule Do, although somewhat verbose, is essentially tautological. It states that, under the protocol $\langle \bar{x}(v) \{P\}, ?\bar{y}(w) \{Q\}, \psi \rangle$, for all $\bar{x}$ such that $P$ holds, it is permitted to execute the expression \textit{“do $v$”}. Furthermore, one can assume that, for some $\bar{y}$ such that $Q$ holds, this expression returns the value $w$ and the remainder of the protocol is $\psi$. This rule encourages reasoning about \textit{“do $v$”} essentially in the same way as one reasons about a function call, that is, in terms of a precondition and postcondition, \textit{not} in terms of jumps and continuations.

4.2.2 \textbf{Shallow Try/With}. Try-With-Shallow (Figure 4) allows reasoning about the construct \textit{“shallow-try $e$ with $h$ \mid r”}. This rule states that, if the expression $e$ conforms to protocol $\Psi$ and postcondition $\Phi$, then, by wrapping it inside an effect handler whose arms are $h$ and $r$, one makes it conform to a different protocol $\Psi'$ and postcondition $\Phi'$. The complexity of this rule is delegated to the auxiliary judgement $\text{shallow-handler } \langle \Psi \} \{\Phi \} \mid h \mid r \langle \Psi' \} \{\Phi' \}$. Intuitively, this judgement means that the handler $h \mid r$ acts as a “puzzle piece” whose “inner shape” is $\langle \Psi \} \{\Phi \}$ and whose “outer shape” is $\langle \Psi' \} \{\Phi' \}$. Let us now examine the definition of this judgement, which appears in Figure 5.

Fig. 4. Selected reasoning rules

---

3The reduction relation may be nondeterministic, due perhaps to the presence of nondeterministic primitive operations, or (in future work) due to built-in support for multiple threads of computation.
This judgement represents what one must prove when one wishes to establish that \( h \mid r \) is a valid effect handler. It is a conjunction of two assertions.

The first conjunct, \( \forall v. \Phi (v, \Psi) \rightarrow ewp (r v) \langle \Psi' \rangle \{(\Phi') \} \wedge match \Psi \) with

\[
\begin{array}{l}
\mid \text{end } \Rightarrow \text{True} \\
\mid ! \tilde{x}(v) \{P\}. ? \tilde{y}(w) \{Q\}. \psi \Rightarrow \\
\hspace{1cm} \forall \tilde{x}, k. \left\{ \begin{array}{l}
Q \rightarrow \\
\text{ewp} (k w) \langle \psi \rangle \{(\Phi') \}
\end{array} \right\} \rightarrow \\
\text{ewp} (h \triangledown k) \langle \Psi' \rangle \{(\Phi') \}
\end{array}
\]

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 law (1), one may assume that \( \Phi (v, \Psi) \) holds; and one must prove that the function application \( r v \) satisfies the outer shape \( \langle \Psi' \rangle \{(\Phi') \} \).

The second conjunct, \( \text{match } \Psi \) with . . . , is a requirement on the first arm, \( h \), which takes control when the monitored expression performs an effect. If \( \Psi \) is end, then this cannot happen (indeed, this protocol forbids all effects), so there is in fact no requirement. If \( \Psi \) is \( ! \tilde{x}(v) \{P\}. ? \tilde{y}(w) \{Q\}. \psi \), then one must prove that the function application \( h \triangledown k \) satisfies the outer shape \( \langle \Psi' \rangle \{(\Phi') \} \).

In so doing, one cannot control \( \tilde{x} \) or \( k \), which are chosen by the monitored expression, but one can assume that the precondition \( P \) holds, and one can assume that the continuation \( k \) is valid. This assumption is expressed by the statement \( \forall \tilde{y}. Q \rightarrow \text{ewp} (k w) \langle \psi \rangle \{(\Phi') \} \), which guarantees that, for an arbitrary choice of \( \tilde{y} \), provided \( Q \) holds, the continuation \( k \) is prepared to accept the reply \( w \) and behaves as per the shape \( \langle \psi \rangle \{(\Phi') \} \), where \( \psi \) is the tail of the original protocol \( \Psi \). This reflects the fact that one interaction (consisting of one reply and one request) has taken place: the continuation can be expected to obey the remainder of the protocol. The absence of a persistence modality \( \Box \) in front of this assumption means that it can be exploited at most once: therefore, the continuation \( k \) can be invoked at most once. This reasoning rule enforces an affine usage of continuations.

There remains to explain why the definition of \( \text{shallow-handler} \) involves an ordinary conjunction, as opposed to a separating conjunction. Requiring a separating conjunction would be sound, but needlessly strong. A separating conjunction of magic wands \( (P \rightarrow P') \ast (Q \rightarrow Q') \) allows both implications to be independently exploited, whereas an ordinary conjunction \( (P \rightarrow P') \land (Q \rightarrow Q') \) 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.

### 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 (§3.2). The rule **Try-With-Deep** (Figure 4) 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 \( deep-handler \left( \Psi \mid \Phi \right) h \mid r \left( \Psi' \mid \Phi' \right) \), whose definition\(^2\) appears in Figure 5. This judgement itself is defined very much in the same way as \( shallow-handler \left( \Psi \mid \Phi \right) h \mid r \left( \Psi' \mid \Phi' \right) \). 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 is \( \forall \gamma. Q \rightarrow ewp \left( k \mid w \right) \left( \gamma \mid \Psi \right) \mid \Phi \). In the case of a deep handler, this assumption takes a different (and, upon close examination, only slightly more complex) form:

\[
\forall \gamma, \Psi'', \Phi''. \left\{ \begin{array}{l}
Q \rightarrow \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\##
we have imposed an affine usage of continuations in the definition of \( ewp \).

An intuitive reading, from left to right, is that if

This is the standard rule \cite[§6.2]{Jung2018}, with one added detail: the two occurrences of

1 is essentially isomorphic to Hinrichsen et al.’s \( iProto \), and the right injection encodes request/reply protocols. The right-hand side of the sum is

therefore, our protocols can be viewed as restricted Actris protocols. Our type of protocols, named

Fig. 6. Definition of protocols

Fig. 6. Definition of protocols

to the remainder of the protocol: it is named \( \Psi’ \) in the outer assertion \( ewp \ e \ (\langle v, \Psi’ \rangle) \ldots \). It is natural, then, that the expression \( N[v] \), which represents the remainder of the computation, should obey the protocol \( \Psi’ \): this is expressed by the inner assertion \( ewp \ N[v] \langle \Psi’ \rangle \{ \Phi \} \).

\textbf{WAND} allows using a magic wand to change a postcondition from \( \Phi_1 \) to \( \Phi_2 \). It is analogous to Jung et al.’s \( \text{wp-WAND} \) \cite[§6.2]{Jung2018} (§2.1) and, as noted by these authors, it implies the traditional consequence rule and frame rule of Separation Logic. The frame rule is written \( P \ast ewp \ e \ (\langle \Psi \rangle \{ \Phi \}) \vdash ewp \ e \ (\langle \Psi \rangle \{ P \ast \Phi \}) \). As noted earlier (§1), this rule does not come for free: it is valid only because we have imposed an affine usage of continuations in the definition of \( ewp \) (§4.1).

Rule \textbf{END} allows using a magic wand to change a protocol from \( \text{end} \) to \( \Psi_2 \). One intuitive reading of this rule, from bottom to top, is that if it is permitted to follow the protocol \( \Psi_2 \), then it is also permitted to perform no effect at all. However, this is actually true only if the postcondition \( \Phi_2 \) allows the remainder of the protocol to be \( \Psi_2 \) itself: this condition is expressed by the assertion \( \Phi_2 \ (\_, \Psi_2) \) in the second premise.

An alternate formulation of \textbf{END} is \( ewp \ e \ (\langle \text{end} \rangle \{ \Phi \}) \vdash ewp \ e \ (\langle \Psi \rangle \{ v, \Psi’ \}) \ast \Phi \ (v, \text{end}) \ast \Psi = \Psi’ \). An intuitive reading, from left to right, is that if \( e \) performs no effect, then it is safe to place \( e \) in a setting where the effect protocol \( \Psi \) is in force; and, when \( e \) terminates, the effect protocol is still \( \Psi \).

The reasoning rules associated with standard programming language constructs (primitive operations; functions; products; sums; references) are not shown; they are standard. To give just one example, the rule for writing a reference is \( \ell \mapsto v \ast (\ell \mapsto w \mapsto \Phi ((\_), \Psi)) \vdash ewp \ (\ell := w) \ (\langle \Psi \rangle \{ \Phi \}) \). This is the standard rule \cite[§6.2]{Jung2018}, with one added detail: the two occurrences of \( \Psi \) indicate that this expression performs no effect.

Because our logic is an instance of Iris, all of the power of Iris is available to it; in particular, ghost state and invariants \cite{Jung2018} can be used, if needed, exactly as usual.

### 4.3 Formal Model of Protocols

Up to this point, we have been using the semi-informal syntax of protocols that was given earlier: \( \Psi := \text{end} \mid ! \ x (v) \{ P \} \ast ? \ y (w) \{ Q \} \). \( \Psi \) (§2.1). We have taken for granted that there is a mathematical space (or, in type-theoretic terms, a type) of protocols and that it is possible in the meta-language to construct and deconstruct protocols. We now give some more details about the manner in which the type of protocols is defined in Iris/Coq.

We draw heavy inspiration from Actris \cite[§5.2]{Hinrichsen2020}. A request/reply protocol

\( ! \ x (v) \{ P \} \ast ? \ y (w) \{ Q \} \). \( \Psi \) is, in essence, the sequential composition of a “send” and a “receive”: therefore, our protocols can be viewed as restricted Actris protocols. Our type of protocols, named \( iEff \) for "effect protocol", is defined in Figure 6. It is a sum \( 1 \ast \ldots \), where the left injection encodes the trivial protocol end, and the right injection encodes request/reply protocols. The right-hand side of the sum is \( \text{Val} \rightarrow (\text{Val} 

-\rightarrow iEff \ast iProp \rightarrow iProp) \rightarrow iProp \). A careful reader could check that \( iEff \) is essentially isomorphic to Hinrichsen et al.’s \( iProto \), suitably specialized and unfolded so as to impose that every “send” be immediately followed by a “receive”.

\footnote{The type-level “later” combinator \( \triangleright \), whose data constructor is next, ensures that this recursive definition is well-formed.}
At an intuitive level, what does this definition mean? If one ignores the “later” combinator and if one reads $T \rightarrow iProp$ as the powerset $\mathcal{P}(T)$, one finds $iEff \cong 1 + \mathcal{P}(Val \times \mathcal{P}(Val \times iEff))$. This suggests, quite naturally, that in a request/reply protocol, there is a choice (made by Player) among a set of values (requests), and that with each such value, comes a choice (made by Opponent) among a set of values (replies) and protocol continuations.

As one may hope or expect, the mathematical space $iEff$ can represent the syntax of protocols; the encoding is given in Figure 6.

The external choice $\Psi_1 + \Psi_2$, which was mentioned and used earlier (§2.1, §2.2) can be defined by cases over $\Psi_1$ and $\Psi_2$, as follows. First, we let $\text{end} + \Psi \triangleq \Psi + \text{end} \triangleq \Psi$. Since the protocol end allows no effect at all, it makes intuitive sense for it to be a unit for external choice. Second, we let $\text{inr} B_1 + \text{inr} B_2 \triangleq \text{inr}(\lambda v. \Phi. B_1 v \Phi \lor B_2 v \Phi)$, where “$\lor$” stands for disjunction in $iProp$. This means that the possibilities offered to Player by the protocol $\Psi_1 + \Psi_2$ are just the union of those offered by $\Psi_1$ and those offered by $\Psi_2$.

4.4 Soundness

It is not difficult, based on our definition of $ewp$ (Figure 3), to prove that our program logic is sound:

THEOREM 4.1. Let $e$ be a closed expression. If $\vdash ewp e \langle \text{end} \rangle \{ \Phi \}$ holds then executing $e$ in an empty initial heap is safe.

If $e$ can be verified under the rules of our program logic, with respect to the protocol “end” and an arbitrary postcondition $\Phi$, 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.

It should be intuitively clear that our reasoning rules allow every continuation to be used at most once (§4.2.2). Yet, our operational semantics does not enforce this policy: using a continuation more than once does not cause a runtime error (§3.4). Therefore, Theorem 4.1 does not formally imply that every continuation is invoked at most once. One could prove that this is the case, at the cost of complicating the operational semantics; we leave this to future work.

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.\footnote{Terminology can be treacherous: a word such as “iterator” means different things to different people. Following Pottier [2017], we use words that we hope are unambiguous.}

A fold is a producer represented as a higher-order function of type ‘$\text{a} \rightarrow \text{unit}$’ $\rightarrow$ unit, where ‘$\text{a}$’ 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 $\rightarrow$ unit) $\rightarrow$ 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 $\rightarrow$ 'a head
  and 'a head = Nil | Cons of 'a * 'a t
```
Thus, a cascade of type 'a Seq.t is a function, a delayed computation. When applied to the value (), it returns either Seq.Nil, which means that the cascade is exhausted, or Seq.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. Because they allow a computation to be suspended and resumed on demand, they offer a simple way of transforming a fold into a cascade. In Multicore OCaml, for instance, this transformation can be implemented once and for all as a third-order function invert, whose code appears in Figure 7.

Fig. 7. Control inversion in Multicore OCaml

```ocaml
let invert (type a) (iter : (a -> unit) -> unit) : a Seq.t =
  let open struct effect Yield : a -> unit end in
  let yield x = perform (Yield x) in
  fun () ->
    match iter yield with
    | effect (Yield x) k -> Seq.Cons (x, continue k)
    | () -> Seq.Nil
```

In Multicore OCaml, continue converts a continuation into an ordinary function.
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.

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. The predicate permitted represents the information that the consumer gains when observing the production of a new element, while the predicate complete represents the information that the consumer obtains when observing that there are no more elements. Furthermore, we parameterize the specifications with an assertion C. 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 8. 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 take us from I us to I (us ++ [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 describe effects in a similar way: we parameterize the specification with a protocol Ψ, which itself is parameterized by the past elements. The initial protocol is Ψ []; if f takes us from Ψ us to Ψ (us ++ [u]), then the final protocol is Ψ us, where us is the final list of elements. This advertises the fact that the effects performed by iter f are the sequence of the effects performed by the calls to f. In other words, 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 8. 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 Seq.Cons (u, k’), in which case the user can rely on the facts that (1) producing u at this point is permitted and (2) k’ is a cascade that can produce the remaining elements, or Seq.Nil, in
which case the user obtains the information that iteration is complete and regains access to the underlying collection, represented by the assertion $C$. (This point will be explained shortly.) The third branch $\bot \Rightarrow \text{False}$ is required because our calculus is untyped. Thus, we must explicitly exclude the possibility that a cascade head might be a value other than $\text{Seq.Nil}$ or $\text{Seq.Cons}(_{-}, _{-})$. The specification of $\text{invert}$, on the last line of Figure 8, states that $\text{invert}$ 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 $\text{iter}$, which is invoked when the first element of the cascade is demanded. It is recovered once the cascade is exhausted.\footnote{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.}

5.4 Verification of $\text{invert}$

The verification of $\text{invert}$ boils down to proving that the $\text{match}$ with $\text{effect}$ expression on line 5 reduces (without effects) to a value $h$ such that $\text{isCascadeHead} h [\ ]$ holds. Naturally, to prove this, we must apply the reasoning rule $\text{Try-With-Deep}$ (Figure 4). The main challenge is to find out with what invariant $I$ and with what protocol $\Psi$ the specification of $\text{iter}$ must be instantiated, in such a way that (1) the function call $\text{iter} \_ \text{yield}$ on line 5 is safe and (2) the code in lines 6–7 forms a correct effect handler.

Let us define $\Psi$ first. Although our specification of $\text{iter}$ allows us to pick a protocol that is parameterized with the past elements, it turns out that this flexibility is not required here. (This point is further discussed below and in §8.) We define “$\Psi$” as the following (recursive) protocol:

\[
\Psi \_ \triangleq ! \ u \ u (u) \ (\text{permitted} (\ u ++ \ [u]) \ * \ I \ u) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \_)
\]

This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.

An aspect of the above definition that may seem odd is that the variable $\ u$ can be instantiated by Player in an arbitrary way, provided $I \ u$ holds. One might have expected instead a parameterized definition $\Psi \ u \triangleq ! \ u \ u (u) \ (\ldots) \ . \ ? \ (((I \ (\ u ++ \ [u]))) \ . \ \Psi \ u)$. This protocol states that when an element $u$ is yielded, $\text{permitted} (\ u ++ \ [u])$ and $I \ u$ hold, and after this effect has been performed, $I \ (\ u ++ \ [u])$ must hold. It may seem as if this repeats information that is already present in the specification of $\text{iter}$. This is true. Building this information into the protocol allows us to exploit it when we prove the correctness of the effect handler.\footnote{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.}

The computation’s view serves as the computation’s loop invariant: we set $I \ u \triangleq \_ \circ \_ \ u \ _\_ \ y$. The handler’s view $\_ \ u \ _\_ \ y$ serves as an assumption in the following lemma, which states that the handler is correct. Because the $\text{deep-handler}$ judgement is recursively defined, this lemma must be proved by Lőb induction. The handler’s view serves as an assumption in an inductive proof: therefore, it also plays the role of a loop invariant.
A Separation Logic for Effect Handlers

1. type 'a status = Done of 'a | Waiting of ('a, unit) continuation list
2. type 'a promise = 'a status ref
3. effect Async : (unit -> 'a) -> 'a promise
4. effect Await : 'a promise -> 'a
5. let async e = perform (Async e)
6. let await p = perform (Await p)
7. let run (main : unit -> unit) : unit =
8.   let q : (unit -> unit) Queue.t = Queue.create () in
9.   let next () = if not (Queue.is_empty q) then Queue.take q () in
10.  let rec fulfill : 'a. 'a promise -> (unit -> 'a) -> unit = fun p e ->
11.     match e() with
12.       | v ->
13.         let Waiting ks = !p in
14.         List.iter (fun k -> Queue.add (fun (_) -> continue k v) q) ks;
15.         p := Done v;
16.         next ()
17.       | effect (Async e') k ->
18.         let p' = ref (Waiting []) in
19.         Queue.add (fun (_) -> continue k p') q;
20.         fulfill p' e'
21.       | effect (Await p') k ->
22.         match !p' with
23.           | Done v -> continue k v
24.           | Waiting ks -> p' := Waiting (k :: ks); next ()
25.       in
26.     fulfill (ref (Waiting [])) main

Fig. 9. A cooperative concurrency library in Multicore OCaml

**Lemma 5.1.** The effect handler on lines 6–7 is correct in the following sense:

\[
\forall us. \frac{\text{deep-handler} \langle \Psi, \_ \rangle \left\{ ((), \_). \exists us. \delta us. \uparrow \right\} \ast \text{complete us} \ast C}{(\text{line } 6)} \\
\frac{\text{end} \left\{ (h, \_). \text{isCascadeHead} h us \right\}}{(\text{line } 7)}
\]

Let us recall that the code that we actually verify is not the Multicore OCaml code of Figure 7: it is a manual transcription of this code into the calculus HH [Anonymous 2020].

Let us also remark that our definition of isCascade does not involve a persistence modality $\Box$. This implies that a cascade can be used at most once. Thus, the rule that a continuation must be invoked at most once is statically enforced by our specification of invert.

**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.
This 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. In this section, we specify and verify a slightly simplified\(^8\) version of it. 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.

### 6.1 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 complete code for this library appears in Figure 9. The (abstract) type `promise` and 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()` terminates with a value `v` (line 12), the promise becomes fulfilled (line 15); the fibers `ks` that are waiting on this promise are retrieved (line 13) and become ready (line 14); and an arbitrary ready fiber is scheduled (line 16).
2. If `e()` performs an effect `Async e'`, then a new promise `p'` is created (line 18); the action of passing this promise to the continuation `k` is considered a new ready fiber (line 19), but is not scheduled immediately; instead, the newly spawned fiber becomes active (line 20). (This is an arbitrary choice; one could do the converse.)
3. 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 23). Otherwise, `k` is added to the set of fibers waiting on `p'`, and an arbitrary ready fiber is scheduled (line 24).

---

\(^8\)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).
A Separation Logic for Effect Handlers

isPromise : Val → (Val → iProp) → iProp persistent(isPromise p ϕ) ∃Ψ

ASYNC

\[\text{ewp } e() \langle Ψ_{conc}\rangle\{(v, _). \ □ ϕ(v)\}\] \[\text{ewp} \ (\text{async } e)\langle Ψ_{conc}\rangle\]
\[\{(p, !Ψ_{conc}). \ \text{isPromise } p \ ϕ\}\]

AWAIT

\[\text{ewp} \ (\text{await } p)\langle Ψ_{conc}\rangle\]
\[\{(v, !Ψ_{conc}). \ □ ϕ(v)\}\]

RUN

\[\text{ewp} \ (\text{run } \text{main})\langle \text{end}\rangle\]
\[\{_. \ True\}\]

This process begins with an application of \text{fulfill} to a dummy promise and to the \text{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).

Multicore OCaml’s type system does not keep track of the effects that a function may perform. If it did, one would see that the argument \(e\) of \text{async} and the function \text{main} may perform Async and Await effects, whereas the functions stored in the queue \(q\), which represent the ready fibers, may not. 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 type 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 10.

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 10). 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. This assertion means that \(p\) is a valid promise and that whatever value \(v\) can be obtained by waiting on this promise satisfies the assertion \(□ \phi(v)\). We write \(□ \phi(v)\), 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 \(□ \phi(v)\) is duplicable.

The specification also declares the existence of a protocol \(Ψ_{conc}\) (Figure 10). 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 \text{main} adheres to this protocol, whereas run main itself obeys the protocol end. 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 \(Ψ_{conc}\) is abstract, except by calling run, there is no way for the user to handle the effects performed by async and await.

\text{ASYNC} states that if the function call \(e()\) returns a value \(v\) that satisfies the assertion \(□ \phi(v)\), then \text{async } e returns a promise \(p\) described by \(isPromise p \ ϕ\). A few aspects of this rule are worth

---

9These functions are captured continuations whose topmost frame is an effect handler. Indeed, this code uses deep handlers.
Paulo Emílio de Vilhena and François Pottier

\[
\Psi_{\text{conc}} \triangleq \Psi_{\text{Async}} + \Psi_{\text{Await}}
\]

\[
\Psi_{\text{Async}} \triangleq ! e (\text{Async } e) \{ \text{ewp } e() (\Psi_{\text{conc}}) \{ (v, \_). \Box \phi(v) \} \}. ? p (p) \{ \text{isPromise } p \phi \}. \Psi_{\text{conc}}
\]

\[
\Psi_{\text{Await}} \triangleq ! p (\text{Await } p) \{ \text{isPromise } p \phi \}. ? v (v) \{ \Box \phi(v) \}. \Psi_{\text{conc}}
\]

**Ready q k** $\triangleq$ PromiseInv $q \rightarrow$ isQueue $q$ (Ready $q$) $\rightarrow$ ewp $k()$ (end) { _ . True }

**PromiseInv q** $\triangleq$

$\exists M. (\__ M )^{\psi_{\text{promise}}}_M *$

\[\forall \{(p, y) \mapsto \phi\} \in M. \exists b. (\__ b )^{\psi}_b *\]

match $b$ with

| false $\Rightarrow$
| $\exists v. p \mapsto \text{Done } v * \Box \phi(v)$
| true $\Rightarrow$
| $\exists ks. p \mapsto \text{Waiting } ks *$

\[\forall k \in ks. \forall v. \phi(v) \rightarrow \text{PromiseInv } q \rightarrow \text{isQueue } q (\text{Ready } q) \rightarrow\]

\[\text{ewp } (k v) \text{ (end) } \{ \_ . \text{True} \} \]

**isPromise p φ** $\triangleq$ $\exists y. \text{isPromise'} p y \phi$

**isPromise' p γ φ** $\triangleq$ $\circ \{ (p, y) \mapsto \phi \}^{\psi_{\text{promise}}}$

Fig. 11. Internal definitions for the verification of the concurrency library

emphasizing. First, in the premise, the expression $e()$ is allowed to follow the protocol $\Psi_{\text{conc}}$. This means that the newly-spawned fiber may call async and await. Second, this rule is an implication $\text{ewp } . . . \rightarrow \text{ewp } . . .$. 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. Finally, in the conclusion, the notation $(p, ! \Psi_{\text{conc}})$. isPromise $p \phi$ is short for $(p, \Psi')$. \( \Psi' = \Psi_{\text{conc}} * \) isPromise $p \phi$. This postcondition guarantees that, after calling async, the effect protocol $\Psi_{\text{conc}}$ is still in force.

**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 if main() is safe to execute, and possibly performs effects as per the protocol $\Psi_{\text{conc}}$, then run main is safe to execute, and performs no effects. The postcondition of run main is True, which means that, 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, as shown in Figure 9, depends on two external modules, namely List and Queue. In Multicore OCaml, these modules are both 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. 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 \( \Psi_{\text{conc}} \), 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 11. Several of them are mutually recursive.

The assertion \( Ready \ q \ k \) describes every fiber \( k \) in the ready queue \( q \). It states that it is safe to execute \( k() \) under the assumptions \( PromiseInv \ q \) and \( isQueue \ q \ (Ready \ q) \) and that this expression performs no effect. The first of these assumptions requires every promise to be well-formed, while the second assumption requires the ready queue to be well-formed and populated with ready fibers. These assumptions also require unique ownership of these data structures, and allow read and write access to them.

The assertion \( PromiseInv \ q \) (still in Figure 11) 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 \( \gamma \) of a ghost cell whose content, a Boolean flag \( b \), indicates whether this promise is unfulfilled; (2) the predicate \( \phi \) 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 do not explain the details. In short, a ghost cell at address \( \gamma_{\text{promise}} \) stores a global map \( M \) whose entries have the form \( (p, \gamma) \mapsto \phi \), and for each such entry, a ghost cell at address \( \gamma \) holds a Boolean flag \( b \). Depending on the value of this flag, an additional requirement is expressed:

1. if \( b \) is false, which means that the promise is fulfilled, then the reference cell at address \( p \) must contain \( Done \ v \), for some value \( v \) such that \( \Box \phi(v) \) holds.
2. otherwise, this cell must contain \( Waiting \ k \), where every continuation \( k \) in the list \( k \) can be safely applied to an arbitrary value \( v \) such that \( \Box \phi(v) \) holds.

Finally, the definition of \( isPromise \ p \ \phi \) (also in Figure 11) 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 \( \text{fulfill} \) (line 10) admits the following specification:

\[
\forall e, p, \gamma, \phi. \begin{cases}
\text{PromiseInv } q \rightarrow isQueue q (\text{Ready } q) \rightarrow \\
\text{isPromise’ } p \gamma \phi \rightarrow \text{True} \}
\end{cases}
\]

In short, this lemma states that if the promise \( p \) is unfulfilled and if \( e() \) produces a value \( v \) that satisfies \( \phi \) then the function call \( \text{fulfill } p \ e \) is safe and performs no effect. Inside the proof of this
lemma, we apply the reasoning rule \textit{Try-With-Deep} (Figure 4) and find an obligation to establish the following \textit{deep-handler} judgement:

\[
\text{deep-handler}\left(\Psi_{\text{conc}}\right) \{(v, \_). \Box \phi(v)\}
\]

(lines 17–24 in Figure 9) \| (lines 12–16 in Figure 9)

\[
\text{end}\{\_\cdot \text{True}\}
\]

The proof of this judgement is carried out by Löb induction. It is straightforward. The assumption that \(p\) is unfulfilled, encoded by the assertion \(\text{\_\cdot \text{true}}\), is exploited to argue that the nonexhaustive case analysis at line 13 cannot fail.

Once more, let us recall that the code that we actually verify is not the Multicore OCaml code of Figure 9: it is a manual transcription of this code into the calculus \textit{HH} [Anonymous 2020].

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. Our protocols (§2) allow this as well. 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 primitive effects such as state and concurrency and user-defined effects and effect handlers.

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 a Separation Logic for a calculus with dynamically-allocated mutable state and an “algebraic” variant of call/cc and throw. Their logic is however unusual in several ways: it uses “large footprint” assertions, therefore has no frame rule; and its rules rely on auxiliary variables that represent snapshots of the heap at various points in time. 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.\footnote{The bind rule states that $\text{wp}_K[e] \{ \Phi \}$ is implied by $\text{wp}_e \{ v . \text{wp}_K[v] \{ \Phi \} \}$ [Jung et al. 2018, §6.2].} Timany and Birkedal define a predicate $\text{wp}$ 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 $\text{clwp}$, 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, the $\text{BIND}$ rule (Figure 4) naturally holds for all neutral evaluation contexts $N$. (We have also proved a version of the bind rule that holds for all evaluation contexts $K$, provided the expression $e$ obeys the protocol end, which guarantees that it performs no effects.) Intuitively, the reason why we are able to reason about an effectful expression without regard for the evaluation context is that we build a protocol, an explicit description of the dialogue between the expression and its context, into the predicate $\text{ewp}$. This form of modularity is a key contribution of our work. Our protocols are inspired by the dependent session protocols found in Actris [Hinrichsen et al. 2020], and the manner in which we model protocols in Coq (§4.3) builds directly on Hinrichsen et al.’s work.

As far as we are aware, no previous work has extended Separation Logic with support for delimited control operators. 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 Separation Logic (presented as a “liquid type system”) where “async” and “await” are primitive operations. 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. The electronic supplement [Anonymous 2020] includes two more case studies, namely simulating exceptions as effects, and implementing a single mutable memory cell as an effect, using a parameterized handler.

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 end 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].
We currently view continuations as affine. However, it would make sense to impose a stronger requirement by viewing continuations as linear. One would then have to use every continuation exactly once, either by invoking it, or by explicitly discarding it. In such a system, exceptions would not be regarded as a special case of effects. Instead, taking inspiration from Multicore OCaml, one would let exceptions and effects coexist, and the primitive operation that discards a continuation (named discontinue in Multicore OCaml) 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 is turn would help programmers ensure that resources, once allocated, are properly deallocated.

Although our logic supports parameterized protocols such as $\Psi_{\text{seq}} \times s$ (§2.2.4), our proof of invert relies on an unparameterized protocol $\Psi_{\_}$ (§5.4) together with a piece of ghost state. This suggests that the very notion of a protocol is perhaps more complex than absolutely necessary. One might get away with a simpler logic where effects are described by fixed signatures as opposed to time-varying protocols. This is perhaps our most pressing avenue for future investigation.

REFERENCES


