You Only Get One Shot: A Separation Logic for One-Shot Undelimited Continuations

PAULO EMÍLIO DE VILHENA, University of Surrey, United Kingdom
XAVIER LEROY, Collège de France, PSL University, France

Capturing the reasoning principles of undelimited continuations in a separation logic has proven challenging: previous logics for `call/cc` abandon either the *frame rule*, the core of separation logic, or the *bind rule*, the key principle to verify programs by parts. We show that by incorporating a *one-shot* restriction, namely that the captured continuations can only be used once, both principles can be recovered in a novel Iris-based separation logic for `call/1cc` and `call/1cc0`, the construct whose semantics incorporates this restriction. The logic enjoys elegant, relatively simple, but yet powerful principles: we show it is sufficient to conduct sophisticated case studies including `call/1cc0`-based implementations of control inversion, cooperative concurrency, and effect handlers similar to Filinski's encoding of `shift`/`reset`. The latter case study further suggests a notion of *context-independent user-defined effects*, which implement a functionality regardless of the control stack. Our results are formalized in Rocq using Iris.

1 Introduction

Control operators such as Scheme's `call-with-current-continuation` (abbreviated `call/cc`), give functional programs the ability to capture, manipulate (as first-class values), and invoke continuations at any point of the program's execution. Many advanced control structures and patterns can be realized using control operators, including resumable exceptions, generators, coroutines, cooperative multithreading, Prolog-style backtracking, and non-blind backtracking. Furthermore, these control structures can be implemented as libraries and used in programs written in direct style; there is no need to write the programs in continuation-passing style (CPS) or in monadic style. In other words, `call/cc` and other control operators pave the way for control structures that are programmed in the language rather than built-in.

The tremendous expressiveness of `call/cc` comes at a cost. First, `call/cc` is low-level and unstructured, resulting in programs that are difficult to write and even harder to read. Some say that `call/cc` is the `goto` of functional languages[1], encouraging "spaghetti" functional code in the same way that `goto` encourages "spaghetti" imperative code. Second, unlike `goto`, `call/cc` is difficult to implement efficiently, requiring clever runtime representations of continuations or specific compiler support.

These difficulties with `call/cc` can be alleviated to some extent by using control operators for delimited continuations (`shift`/`reset`) or effect handlers for user-defined effects, which provide more program structure than `call/cc`. Nevertheless, the original `call/cc` operator remains an interesting subject of study in programming-language research, owing to its simple operational semantics and CPS transformation. Insights gained through the study of `call/cc` can also inspire new perspectives on effect handlers, as demonstrated in §8.

Program logics are an effective way to better understand low-level, unstructured language constructs. Spaghetti code becomes more understandable once specified using logical assertions, and safer once verified using an appropriate program logic. As demonstrated by Turing [1949] and Floyd [1967], it is possible to reason about flowcharts by annotating them with assertions at various program points. As demonstrated by Clint and Hoare [1972], it is possible to reason about `goto` statements in Algol-like languages using Hoare logic and assertions on `goto` labels.

[1] One of them, at least; others say that general recursion is the `goto` of functional languages.

Authors' Contact Information: Paulo Emílio de Vilhena, p.devilhena@surrey.ac.uk, University of Surrey, Guildford, United Kingdom; Xavier Leroy, xavier.leroy@college-de-france.fr, Collège de France, PSL University, Paris, France.

In the same spirit, several program logics have been proposed to reason about `call/cc` in functional languages, such as those of Berger [2009], Crolard and Polonowski [2012], Delbianco and Nanevski [2013], Timany and Birkedal [2019], and de Vilhena [2026]; see §9.1 for a review. These logics capture interesting reasoning principles for `call/cc`. However, they are not fully satisfactory: they abandon either the *frame rule*, the core of separation logic [Reynolds 2002], or the *bind rule*, the key principle for verifying programs by parts.

In this paper, we present a separation logic for the `call/1cc` control operator of Bruggeman et al. [1996]. This operator is a variant of `call/cc` that captures *one-shot* undelimited continuations, *i.e.* continuations that can be invoked at most once. Initially motivated by implementation considerations, one-shot continuations are an excellent fit for separation logic, where they act as affine resources. With the notable exception of backtracking, most uses of `call/cc` naturally fit the one-shot restriction and can be specified and verified using our separation logic.

Our logic is defined within the *Iris* framework [Jung et al. 2018] and mechanized using the Rocq prover. It supports the unrestricted bind and frame rules as well as higher-order mutable state, thus enabling modular reasoning in the presence of continuations that are stored in mutable references, a ubiquitous feature in most applications of `call/cc`. Our logic also benefits from Iris's support for *ghost state* and *guarded recursion*. We show that this logic is sufficient to specify and verify nontrivial libraries that leverage one-shot undelimited continuation to implement cooperative multithreading, control inversion, and effect handlers.

The remainder of this paper is organized as follows. The next three sections provide background material: §2 on `call/cc` and its typical uses, §3 on the Hoare logic for `call/cc` proposed by de Vilhena [2026], and §4 on `call/1cc`, the one-shot restriction of `call/cc` of Bruggeman et al. [1996]. Section 5 describes our separation logic for `call/1cc` and its variant `call/1cc0`. Section 6 describes the formalization and adequacy proof of this logic in the Iris framework. Section 7 uses this Iris-encoded logic to verify two classic uses of `call/cc`: cooperative multithreading and control inversion. Section 8 derives logical rules for two types of effect handling by implementing them with `call/1cc`. Related work is discussed in §9 and followed by concluding remarks in §10.

2 Call/cc by examples

We start with a quick reminder on the `call/cc` control operator. The expression `call/cc` (λk. e) evaluates e with k bound to the *undelimited continuation* of the `call/cc` expression, which, from a programming perspective, corresponds to the program point where this expression occurs. If e returns normally, then e's value is that of the whole `call/cc` expression, and the execution proceeds normally. If, during the evaluation of e or at any later time, the continuation k is *invoked* with a value v, which we write `throw` k v, the execution proceeds as if the `call/cc` expression had returned v. For example, the function

```
λn. call/cc (λk. - (if n < 0 then n else throw k n))
```

computes the absolute value of n. Indeed, if n is negative, the body of the `call/cc` returns `-n` normally. Otherwise, the body invokes k with n, thus jumping over the computation of the opposite and returning early with value n.

Here is another example showing the captured continuation being invoked after `call/cc` returned:

```
match call/cc (λk. (1, Some k)) with
| (x, Some k) -> throw k (x + 1, None)
| (x, None)   -> x
```

The pattern-matching that is the continuation of the `call/cc` expression is executed twice, the first time with the value `(1, Some k)` returned normally, the second time with the value `(2, None)` sent by `throw`.

An important use of `call/cc` is to implement custom control structures as libraries. As an example, here is a simple implementation of cooperative multithreading, where multiple threads interleave execution at yield points:

```
let ready = Queue.create ()
let restart () =
  if Queue.is_empty ready then () else Queue.take ready ()
let spawn f =
  Queue.add ready (λ(). f (); restart ())
let yield () =
  call/cc (λk. Queue.add ready (throw k); restart ())
```

The queue `ready` contains threads (represented as `unit → unit` functions) that are suspended and ready to run. The operation `spawn f` creates a new thread by enqueuing `f`. A running thread can yield execution to a ready thread by calling `yield`. It does so by enqueuing the continuation it obtains via `call/cc`, thus effectively suspending its own execution. The final call to `restart` then transfers control to a ready thread.

Another important application of `call/cc` is to implement nondeterminism using backtracking:

```
let choice_points = Stack.create ()
let fail () =
  let k = Stack.pop choice_points in k ()
let either f g =
  call/cc (λk. Stack.push (λ(). throw k (g ())) choice_points; f ())
```

The stack `choice_points` stores *choice points*: continuations expecting unit values. The function `fail` aborts the current computation and restarts the most recent choice point. Nondeterministic choice is performed by `either f g`, which returns `f ()` initially, but pushes a choice point that will cause `g ()` to be evaluated if `f` fails.

Our goal in this paper is to develop appropriate program logics for specifying library functions such as `spawn`, `yield`, `either` and `fail`, proving their correctness, and verifying user code that uses these library functions. In particular, we wish to derive reasoning principles for undelimited continuations in *separation logic* [Reynolds 2002], and specifically in *Iris* [Jung et al. 2018], so as to (1) enable modular reasoning in the presence of higher-order mutable state, a ubiquitous feature in most applications of `call/cc` (and its variants `call/1cc` and `call/1cc0`), and (2) leverage Iris's support for *ghost state* and *guarded recursion*, as required by our case studies (§7).

3 A Hoare logic for call/cc

A starting point for this paper is the program logic for `call/cc` of de Vilhena [2026], itself based on the Maze logic of de Vilhena [2022, Chapter 6]. The logic applies to a small, ML-style language of call-by-value functions, mutable references, and the `call/cc` and `throw` control operators:

Expressions:	$e ::= \mathtt{x} \mid \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 \mid e_1; e_2$	computations
	$\mid \lambda \mathtt{x}.\, e \mid e_1\ e_2$	functions
	$\mid \mathtt{ref}\ e \mid\ !e \mid e_1 := e_2$	references
	$\mid \mathtt{call/cc}\ (\lambda \mathtt{k}.\, e) \mid \mathtt{throw}\ e_1\ e_2$	control

$$\textsc{Call/cc}\ \frac{\forall k,\ \{\, P * \mathit{isCont}\ k\ Q \,\}\ e[\mathtt{x} := k]\ \{\, Q \,\}}{\{\, P \,\}\ \mathtt{call/cc}\ (\lambda \mathtt{x}.\, e)\ \{\, Q \,\}} \qquad \textsc{Throw}\ \frac{}{\{\, \mathit{isCont}\ k\ Q * Q\ v \,\}\ \mathtt{throw}\ k\ v\ \{\, \_.\, \mathit{False} \,\}}$$

$$\textsc{Return}\ \frac{P \vdash Q\ v}{\{\, P \,\}\ v\ \{\, Q \,\}} \qquad \textsc{Bind}\ \frac{\{\, P \,\}\ e_1\ \{\, R \,\} \qquad \forall v,\ \{\, R\ v \,\}\ e_2[\mathtt{x} := v]\ \{\, Q \,\}}{\{\, P \,\}\ \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2\ \{\, Q \,\}} \qquad \textsc{App}\ \frac{\{\, P \,\}\ e[\mathtt{x} := v]\ \{\, Q \,\}}{\{\, P \,\}\ (\lambda \mathtt{x}.\, e)\ v\ \{\, Q \,\}}$$

$$\textsc{Consequence}\ \frac{P \vdash P' \qquad \{\, P' \,\}\ e\ \{\, Q' \,\} \qquad \forall v,\ Q'\ v \vdash Q\ v}{\{\, P \,\}\ e\ \{\, Q \,\}} \qquad \textsc{Alloc}\ \frac{}{\{\, P \,\}\ \mathtt{ref}\ v\ \{\, r.\, r \mapsto v * P \,\}}$$

$$\textsc{Read}\ \frac{}{\{\, r \mapsto v * P \,\}\ !r\ \{\, y.\, y = v * r \mapsto v * P \,\}} \qquad \textsc{Write}\ \frac{}{\{\, r \mapsto \_ * P \,\}\ r := v\ \{\, \_.\, r \mapsto v * P \,\}}$$

Fig. 1. Hoare logic rules for `call/cc` (multi-shot continuations).

Figure 1 shows a set of Hoare logic rules for this language. The rules define triples $\{\, P \,\}\ e\ \{\, Q \,\}$ where e is an expression, P an assertion on the initial state, and Q a function from the value of e to an assertion on the final state.

The rules for computations, functions and references are standard. The rules for `call/cc` and `throw` use an assertion $\mathit{isCont}\ k\ Q$ stating that k is a valid continuation and can be safely applied to a value v provided that the assertion $Q\ v$ holds. Continuations captured by `call/cc` are undelimited; therefore, unlike regular functions, continuations never return and are specified only by their preconditions.

The (Call/cc) rule states that the precondition Q of the continuation k captured by $e' = \mathtt{call/cc}\ (\lambda \mathtt{x}.\, e)$ is the postcondition of e'. This ensures that $Q\ v$ holds if $\mathtt{throw}\ k\ v$ is executed, causing v to be returned as the value of e'. Moreover, Q must also be the postcondition of e, since $Q\ v$ must also hold if e terminates normally with value v.

The (Throw) rule $\{\, \mathit{isCont}\ k\ Q * Q\ v \,\}\ \mathtt{throw}\ k\ v\ \{\, \_.\, \mathit{False} \,\}$ states that a continuation k can be invoked with value v provided that it is a valid continuation and its precondition Q is satisfied by v. Since `throw` never returns normally, its postcondition can be false.

The (Call/cc) and (Throw) rules are reminiscent of Clint and Hoare's rule for Algol-style `goto` statements [Clint and Hoare 1972]:

$$\frac{\{\, R \,\}\ \mathtt{goto}\ L\ \{\, \mathit{False} \,\} \vdash \{\, P \,\}\ s_1\ \{\, R \,\} \qquad \{\, R \,\}\ \mathtt{goto}\ L\ \{\, \mathit{False} \,\} \vdash \{\, R \,\}\ s_2\ \{\, Q \,\}}{\{\, P \,\}\ \mathtt{begin}\ s_1;\ L : s_2\ \mathtt{end}\ \{\, Q \,\}}$$

The statements s_1 and s_2 that can jump to label L are verified under the assumption $\{\, R \,\}\ \mathtt{goto}\ L\ \{\, \mathit{False} \,\}$, *i.e.* that they can execute `goto` L as long as the assertion R holds. Here, R describes the state at label L: it is both the postcondition of s_1 and the precondition of s_2. In the Hoare logic of de Vilhena [2026], the assertion $\mathit{isCont}\ k\ Q$ roughly plays the role of the hypothetical triple $\{\, R \,\}\ \mathtt{goto}\ L\ \{\, \mathit{False} \,\}$. This is consistent with the encoding of Algol's `goto` using the J control operator proposed by Landin [1965].

The logic in Figure 1 uses the separating conjunction '$*$' in order to formulate some rules conveniently. Nonetheless, this is not a proper separation logic, because it lacks the frame rule "if $\{\, P \,\}\ e\ \{\, Q \,\}$ then $\{\, P * R \,\}\ e\ \{\, v.\, Q\ v * R \,\}$". As shown in de Vilhena [2026], the frame rule is

generally unsound in this logic, and can only be used for expressions e that involve neither `call/cc` nor `throw`.

The unsoundness of the frame rule can be traced back to the fact that the *isCont k Q* assertion is *persistent* in de Vilhena [2026]: it can be used multiple times, which makes it possible for a `call/cc` expression to return multiple times. De Vilhena [2026]'s example of unsoundness crucially relies on `call/cc` returning twice. This suggests that if continuations were *affine*, *i.e.* restricted to be usable at most once, *isCont k Q* would not be persistent, and the unrestricted frame rule might hold. The remainder of this paper explores this idea.

4 One-shot continuations

Bruggeman et al. [1996] introduced `call/1cc`, a restriction of `call/cc` where continuations are *one-shot* and can be invoked at most once. Crucially, returning normally from the body of the `call/1cc` counts as one invocation of the continuation. Consider a `call/1cc` implementation of the absolute-value function of §2:

```
λn. call/1cc (λk. - (if n < 0 then n else throw k n))
```

This is a valid use of `call/1cc`: the continuation is invoked exactly once, either explicitly via `throw k n` when $n \geq 0$, or implicitly because the body returns `n` when $n < 0$.

Similarly, the cooperative multithreading example uses continuations linearly and can be implemented with `call/1cc`:

```
let restart () =
  if Queue.is_empty ready then () else Queue.take ready ()
let yield () =
  call/1cc (λk. Queue.add ready (throw k); restart ())
```

The continuation `k` of `yield ()` captured by `call/1cc` is used exactly once, in `restart`, when the partial application `throw k` reaches the head of the `ready` queue. The body of the `call/1cc` never returns normally, since `restart` does not return normally when the `ready` queue is not empty.

More generally, many uses of `call/cc` to implement advanced control structures fit the linearity restriction of `call/1cc`. This includes restartable exceptions, various flavors of coroutines, Python-style generators, and other forms of control inversion.

The few examples that cannot be implemented with `call/1cc` typically involve backtracking. Consider again the nondeterminism example of §2:

```
let fail () =
  let k = Stack.pop choice_points in k ()
let either f g =
  call/cc (λk. Stack.push (λ(). throw k (g ())) choice_points; f ())
```

The continuation of `either f g` is invoked a first time when `f ()` terminates and its value is returned by the body of `call/cc`. If `fail` is called later, the continuation is invoked a second time, during the execution of the function `λ(). throw k (g ())` that was pushed on the choice points stack. With `call/1cc`, it is not possible to return normally from the body of the `call/1cc`, then invoke the captured continuation later: this counts as two uses of the continuation.[2]

The main motivation for studying `call/1cc` is that it supports a simple implementation based on mutable stacks; no copying of stack fragments is needed, unlike with stack-based implementations of `call/cc` [Bruggeman et al. 1996]. In a nutshell, the continuation of `call/1cc` $(\lambda x.\, e)$ is the stack

[2]The one-shot continuations studied by Friedman and Haynes [1985] explicitly allow this pattern of returning normally then invoking the captured continuation once. We discuss this difference with `call/1cc` in §7.3 and §9.3.

$$\textsc{Call/1cc0}\quad \frac{\forall k,\ \{\!\{ P * \mathit{isCont}_1\ k\ Q \}\!\}\ e[\mathtt{x} := k]\ \{\!\{ \_.\ \mathit{False} \}\!\}}{\{\!\{ P \}\!\}\ \mathtt{call/1cc0}\ (\lambda \mathtt{x}.\ e)\ \{\!\{ Q \}\!\}}$$

$$\textsc{Call/1cc}\quad \frac{\forall k,\ \{\!\{ P * \mathit{isCont}_1\ k\ Q \}\!\}\ e[\mathtt{x} := k]\ \{\!\{ v.\ Q\ v * \mathit{isCont}_1\ k\ Q \}\!\}}{\{\!\{ P \}\!\}\ \mathtt{call/1cc}\ (\lambda \mathtt{x}.\ e)\ \{\!\{ Q \}\!\}}$$

$$\textsc{Throw}\quad \frac{}{\{\!\{ \mathit{isCont}_1\ k\ Q * Q\ v \}\!\}\ \mathtt{throw}\ k\ v\ \{\!\{ \_.\ \mathit{False} \}\!\}}$$

$$\textsc{Return}\quad \frac{P \vdash Q\ v}{\{\!\{ P \}\!\}\ v\ \{\!\{ Q \}\!\}} \qquad \textsc{Bind}\quad \frac{\{\!\{ P \}\!\}\ e_1\ \{\!\{ R \}\!\} \qquad \forall v,\ \{\!\{ R\ v \}\!\}\ e_2[\mathtt{x} := v]\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2\ \{\!\{ Q \}\!\}} \qquad \textsc{App}\quad \frac{\{\!\{ P \}\!\}\ e[\mathtt{x} := v]\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ (\lambda \mathtt{x}.\ e)\ v\ \{\!\{ Q \}\!\}}$$

$$\textsc{Frame}\quad \frac{\{\!\{ P \}\!\}\ e\ \{\!\{ Q \}\!\}}{\{\!\{ P * R \}\!\}\ e\ \{\!\{ v.\ Q\ v * R \}\!\}} \qquad \textsc{Consequence}\quad \frac{P \vdash P' \qquad \{\!\{ P' \}\!\}\ e\ \{\!\{ Q' \}\!\} \qquad \forall v,\ Q'\ v \vdash Q\ v}{\{\!\{ P \}\!\}\ e\ \{\!\{ Q \}\!\}}$$

$$\textsc{Alloc}\quad \frac{}{\{\!\{ \mathit{True} \}\!\}\ \mathtt{ref}\ v\ \{\!\{ r.\ r \mapsto v \}\!\}} \qquad \textsc{Read}\quad \frac{}{\{\!\{ r \mapsto v \}\!\}\ !\, r\ \{\!\{ y.\ y = v * r \mapsto v \}\!\}}$$

$$\textsc{Write}\quad \frac{}{\{\!\{ r \mapsto \_ \}\!\}\ r := v\ \{\!\{ \_.\ r \mapsto v \}\!\}}$$

Fig. 2. Separation logic rules for `call/1cc` and `call/1cc0` (one-shot continuations).

S_1 where this expression evaluates. A new stack S_2 is allocated by `call/1cc`; e is evaluated on this new stack S_2, with `k` bound to a pointer to stack S_1. When e returns normally or when `throw` $k\ v$ is evaluated, the current stack is discarded and execution continues on the original stack S_1 pointed to by k. Because of the one-shot restriction, k will never be invoked again, hence the stack S_1 can be modified in place in the following execution steps; there is no need to copy parts of S_1.

In studying program logics for `call/1cc`, we found it useful to introduce `call/1cc0`, an even more restricted form of `call/cc`. In `call/1cc0` $(\lambda \mathtt{x}.\ e)$, not only the continuation k can be invoked explicitly at most once, but the body e must not return normally: it can only terminate by explicitly invoking a continuation, be it k or another previously-captured continuation. No expressiveness is lost, because `call/1cc` can be expressed trivially in terms of `call/1cc0`:

$$\mathtt{call/1cc}\ (\lambda \mathtt{k}.\ e)\ \equiv\ \mathtt{call/1cc0}\ (\lambda \mathtt{k}.\ \mathtt{throw}\ \mathtt{k}\ e)$$

Moreover, some uses of `call/1cc` naturally fit the `call/1cc0` discipline. This is the case for the cooperative multithreading example above, where the body of the `call/1cc` invocation in `yield` never terminates normally, but always terminates by invoking a continuation corresponding to a ready thread.

5 A separation logic for call/1cc

We now describe the separation logic for one-shot continuations that is the main result of this paper. Figure 2 shows the rules for this logic. They are quite close to the Hoare logic rules for `call/cc` shown in Figure 1, with some crucial differences explained next. We write separation logic triples $\{\!\{ P \}\!\}\ e\ \{\!\{ Q \}\!\}$, to distinguish them from the previous Hoare logic triples $\{ P \}\ e\ \{ Q \}$.

The first difference is the use of the $\mathit{isCont}_1\ k\ Q$ assertion. Like $\mathit{isCont}\ k\ Q$, it says that k is a valid continuation with precondition Q. However, $\mathit{isCont}_1\ k\ Q$ is not persistent and cannot be duplicated. This reflects the one-shot nature of continuations. In particular, in the (Throw) rule, $\texttt{throw}\ k\ v$ consumes the precondition $\mathit{isCont}_1\ k\ Q$ and does not give it back, making it impossible to invoke the continuation k a second time later.

The non-persistent nature of $\mathit{isCont}_1\ k\ Q$ also means that it expresses ownership of resources associated with the continuation k, just like a points-to assertion $\ell \mapsto v$ expresses ownership of the memory area at address ℓ. This makes our logic a proper separation logic, with an unrestricted frame rule (Frame). Thinking in terms of a stack-based implementation of `call/1cc`, one of the resources owned by $\mathit{isCont}_1\ k\ Q$ is the stack that k points to and on which the continuation will run when invoked. Unique ownership guarantees that no other computation is modifying this stack. Thinking more abstractly in program-proof terms, the resources owned by $\mathit{isCont}_1\ k\ Q$ can also include resources that the continuation needs to run safely beyond those provided by the precondition Q. This is formally reflected by the following entailment:

$$\mathit{isCont}_1\ k\ (v.\, P * Q\ v) * P\ \vdash\ \mathit{isCont}_1\ k\ Q$$

In more words: if the continuation k needs two sets of resources P and $Q\ v$, and if the resources P are available now, they can be transferred to the continuation, weakening its precondition to just Q. Another, equivalent formulation of this property is to say that $\mathit{isCont}_1\ k\ Q$ is antimonotonic in Q:

$$(\forall v,\ Q'\ v \mathrel{-\!\!*} Q\ v)\ \vdash\ \mathit{isCont}_1\ k\ Q \mathrel{-\!\!*} \mathit{isCont}_1\ k\ Q'$$

With these properties of isCont_1 assertions in mind, the rules in Figure 2 should make sense. The (Throw) rule $\{\!\{ \mathit{isCont}_1\ k\ Q * Q\ v \}\!\}\ \texttt{throw}\ k\ v\ \{\!\{ \_.\, \mathit{False} \}\!\}$ looks exactly like the Hoare logic (Throw) rule in Figure 1, but now it also expresses that the continuation k itself as well as the extra resources $Q\ v$ it needs are consumed by the `throw` operation and cannot be used elsewhere.

In the rule (Call1cc0), the body e of $\texttt{call/1cc0}\ (\lambda \texttt{x}.\, e)$ is verified with the always false post-condition, to enforce that e never returns normally. The rule (Call1cc) can be deduced from rules (Call1cc0) and (Throw), and from the encoding of $\texttt{call/1cc}\ (\lambda \texttt{k}.\, e)$ as $\texttt{call/1cc0}\ (\lambda \texttt{k}.\, \texttt{throw k}\ e)$. The premise of rule (Call1cc), namely $\{\!\{ P * \mathit{isCont}_1\ k\ Q \}\!\}\ e\ \{\!\{ v.\, Q\ v * \mathit{isCont}_1\ k\ Q \}\!\}$, can also be read as stating that if e terminates with value v, it must not only ensure the postcondition $Q\ v$ but also give the assertion $\mathit{isCont}_1\ k\ Q$ back. This proves that e did not store the continuation k in mutable storage or include it in the return value v.

6 Iris formalization

We now explain how the logic for one-shot continuations from Figure 2 can be formalized in Iris. The formalization also includes multi-shot continuations and the logic from Figure 1.

6.1 Context-based operational semantics for continuations

Iris provides a default language-agnostic program-logic construction on which we rely. The first step to obtain this program logic is to make it language specific. To do so it suffices to define a set of *reduction rules* formalizing the operational semantics of programs (necessary in Iris for its internal definition of the *weakest precondition*). Apart from making the paper self-contained, a presentation of the operational semantics helps clarify the difference between all variants of `call/cc` and

explain some design choices that strengthen the guarantees of the logic. Our semantics follows the *reductions under contexts* approach popularized by Wright and Felleisen [1994].

The `call/cc` examples from §2 suggest that, from a programming perspective, an undelimited continuation corresponds to a program point. In a Wright-Felleisen-style operational semantics, program points are identified by *evaluation contexts*. More specifically, an evaluation context K identifies the position of the next program expression e to be evaluated. The definition of evaluation contexts therefore determines the order of evaluation:

$$\begin{aligned}\text{Evaluation contexts:}\quad K ::= {} & \bullet \mid \texttt{let x} = K \texttt{ in } e \mid e\ K \mid K\ v \mid K\texttt{;}\ e \\ & \mid \texttt{ref}\ K \mid \texttt{!}K \mid e := K \mid K := v \\ & \mid \texttt{throw}\ e\ K \mid \texttt{throw}\ K\ v\end{aligned}$$

Here, for example, the inclusion of $e\ K$ and $K\ v$ in the syntax of contexts determines an evaluation order whereby function arguments are evaluated from right to left.

We write $K[e]$ to mean the program preserving the structure of K and where $\bullet$, the empty context, is substituted with e. It is in this sense that K can be seen as the position where e occurs in $K[e]$. Because K is needed to give operational meaning to continuations, our reduction rules will keep this global view whereby a program is split into an evaluation context K and an expression e about to be evaluated. Here are two illustrative cases capturing beta reduction in this style:

$$\begin{aligned}K[(\lambda \texttt{x}.\, e)\ v] &\ \longrightarrow\ K[e[\texttt{x} := v]] \\ K[\texttt{let x} = v \texttt{ in } e] &\ \longrightarrow\ K[e[\texttt{x} := v]]\end{aligned}$$

The reduction rule for `call/cc` makes it clear that a continuation corresponds to the reification of the surrounding evaluation context K as the value `kont` K:

$$K[\texttt{call/cc}\ (\lambda \texttt{k}.\, e)] \ \longrightarrow\ K[e[\texttt{k} := \texttt{kont}\ K]]$$

The value `kont` K works as a checkpoint storing the position where the `call/cc` expression occurred. This checkpoint can be restored by invoking `kont` K with a value:

$$K'[\texttt{throw}\ (\texttt{kont}\ K)\ v] \ \longrightarrow\ K[v]$$

One-shot continuations, on the other hand, are represented as values of the form `cont` $\ell\ K$, which, in addition to the reified context K, carry a memory location ℓ. This location is used to implement a dynamic one-shot check: it is allocated when the continuation is created and deallocated when the continuation is first invoked. This dynamic check strengthens the guarantees of the logic: using a one-shot continuation twice triggers a double deallocation error; therefore, by ruling out memory errors, the logic also rules out violations of the one-shot policy. In the stack-based implementation of `call/1cc` outlined in §4, the location ℓ can be viewed as the stack associated with the one-shot continuation.

To specify the action of programs on the state, we augment reduction rules with terms σ and σ' to denote the state of the heap before and after the evaluation of an expression e as follows: $K[e]\ /\ \sigma \longrightarrow K'[e']\ /\ \sigma'$. Their omission (for example, in the previous rules) means that the state is left unchanged. Because one-shot continuations interact with the store, the state is apparent in the following rules for `call/1cc0` and `throw`:

$$\begin{aligned}K[\texttt{call/1cc0}\ (\lambda \texttt{k}.\, e)]\ /\ \sigma\ &\longrightarrow\ e[\texttt{k} := \texttt{cont}\ \ell\ K]\texttt{;}\ \texttt{assert false}\ /\ \sigma[\ell \mapsto ()] && (\ell \notin \sigma) \\ K'[\texttt{throw}\ (\texttt{cont}\ \ell\ K)\ v]\ /\ \sigma\ &\longrightarrow\ K[v]\ /\ \sigma \setminus \ell && (\ell \in \sigma)\end{aligned}$$

Apart from the one-shot nature of `cont` $\ell\ K$ and from the dynamic check explained before, a key difference between the rules for `call/cc` and `call/1cc0` is that, whereas, with `call/cc`, the body expression e is executed under K, with `call/1cc0`, this expression is evaluated in a new context $\bullet\texttt{;}\ \texttt{assert false}$. In combination with the one-shot check, this means K is truly affine: a

Weakest precondition.

$$\begin{aligned} wp/cc\ e\ \{\, Q \,\} &= \forall K,\ (\Box \forall v,\ Q\ v \mathrel{-\!\!*} wp\ K[v]\ \{\, \_.\ True \,\}) \mathrel{-\!\!*} wp\ K[e]\ \{\, \_.\ True \,\} \\ wp/cc\ e\ \{\!\{\, Q \,\}\!\} &= \forall R,\ R \mathrel{-\!\!*} wp/cc\ e\ \{\, y.\ Q\ y * R \,\} \end{aligned}$$

Triples.

$$\begin{aligned} \{\, P \,\}\ e\ \{\, Q \,\}_1 &= P \mathrel{-\!\!*} wp/cc\ e\ \{\, Q \,\} \\ \{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}_1 &= P \mathrel{-\!\!*} wp/cc\ e\ \{\!\{\, Q \,\}\!\} \\ \{\, P \,\}\ e\ \{\, Q \,\} &= \Box\{\, P \,\}\ e\ \{\, Q \,\}_1 \\ \{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\} &= \Box\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}_1 \end{aligned}$$

Continuation predicates.

$$\begin{aligned} isCont\ k\ Q &= \forall v,\ \{\!\{\, Q\ v \,\}\!\}\ \texttt{throw}\ k\ v\ \{\!\{\, \_.\ False \,\}\!\} \\ isCont_1\ k\ Q &= \forall v,\ \{\!\{\, Q\ v \,\}\!\}\ \texttt{throw}\ k\ v\ \{\!\{\, \_.\ False \,\}\!\}_1 \end{aligned}$$

Fig. 3. Definition of weakest precondition, triples, and continuation predicates.

program of the form $K[v]$ occurs at most once. Because e is evaluated in a new context, normal termination is now a programming error. To simulate normal termination (as in the implementation of `call/1cc`), the programmer must invoke the captured continuation with its result. The command `assert false` is another dynamic check to strengthen the guarantees of the logic in this aspect: normal termination triggers a safety failure, therefore, by enforcing safety, the logic also enforces a continuation is correctly invoked to restore the initial context.[3]

6.2 Defining weakest preconditions and triples

With the formal characterization of the semantics through reduction rules (and with some routine ghost-state setup to model points-to assertions à la Iris), we obtain a notion of weakest precondition $wp\ e\ \{\, Q \,\}$ satisfying the following rules (among others):

$$\textsc{WpCall/cc}\quad \frac{wp\ K[e[\texttt{k} := \texttt{kont}\ K]]\ \{\, Q \,\}}{wp\ K[\texttt{call/cc}\ (\lambda \texttt{k}.\ e)]\ \{\, Q \,\}} \qquad \textsc{WpKont}\quad \frac{wp\ K[v]\ \{\, Q \,\}}{wp\ K'[\texttt{throw}\ (\texttt{kont}\ K)\ v]\ \{\, Q \,\}}$$

$$\textsc{WpCall/1cc0}\quad \frac{\forall \ell,\ \ell \mapsto ()\ \mathrel{-\!\!*}\ wp\ (e[\texttt{k} := \texttt{cont}\ \ell\ K];\ \texttt{assert false})\ \{\, Q \,\}}{wp\ K[\texttt{call/1cc0}\ (\lambda \texttt{k}.\ e)]\ \{\, Q \,\}} \qquad \textsc{WpCont}\quad \frac{wp\ K[v]\ \{\, Q \,\} \qquad \ell \mapsto ()}{wp\ K'[\texttt{throw}\ (\texttt{cont}\ \ell\ K)\ v]\ \{\, Q \,\}}$$

The assertion $wp\ e\ \{\, Q \,\}$ means e is *safe*: it either diverges or terminates with a value v that satisfies the postcondition Q. The above principles rephrase the reduction rules from §6.1 in this language: for example, to show $K[\texttt{call/cc}\ (\lambda \texttt{k}.\ e)]$ is safe, it suffices to show that, after one execution step, $K[e[\texttt{k} := \texttt{kont}\ K]]$ is safe. Because the one-shot discipline is dynamically enforced using memory allocation and deallocation, Rules (WpCall/1cc0) and (WpCont) respectively provide and ask for the unique ownership of the reference ℓ in $\texttt{cont}\ \ell\ K$.

By default, in Iris, a Hoare triple for a program e with precondition P and postcondition Q is defined on top of wp as $\Box(P \mathrel{-\!\!*} wp\ e\ \{\, Q \,\})$. Following this recipe, we derive the same program

[3]In the semantics of $\texttt{call/1cc0}\ (\lambda \texttt{k}.\ e)$, we assume e can either terminate normally or invoke a continuation. A realistic implementation must take into account other forms of control flow (for example, exceptions and user-defined effects). Extending our approach with delimited-control operators is a direction for future work (§10).

logic as Timany and Birkedal [2019] (except that they target a language without `call/1cc0` and without one-shot continuations in general). As shown by Timany and Birkedal, however, this logic does not validate the bind rule.

Without the bind rule, one can never focus on a subexpression e of a complete program $K[e]$: the logic keeps a global view of programs and limits reasoning to the operational-style rules for *wp*. To address this limitation, Timany and Birkedal introduce the *context-local weakest precondition*:

$$\mathit{clwp}\ e\ \{\, Q \,\} \ =\ \forall K\ Q',\ (\forall v,\ Q\ v \mathrel{-\!\!*} \mathit{wp}\ K[v]\ \{\, Q' \,\}) \mathrel{-\!\!*} \mathit{wp}\ K[e]\ \{\, Q' \,\}$$

The idea is to abstract over the evaluation context K in which e occurs, and to assume that K can be filled with any return value v that satisfies Q. This construction is similar to the semantic technique of *biorthogonality* [Pitts and Stark 1999] and can be seen as a way to close *wp* under the bind rule. Indeed, a logic build on top of *clwp* now validates the bind rule. However, this construction does not improve reasoning support for `call/cc`: the user must unfold the definition of *clwp* and fall back to the operational-style rules of *wp*.

We notice that, with small changes to the definition of *clwp*, we can introduce a notion of weakest precondition that can be used as the foundation of the program logic from Figure 1, thus validating at the same time the bind rule and reasoning principles for `call/cc`. We call it *weakest precondition with current continuation*, written $\mathit{wp/cc}\ e\ \{\, Q \,\}$. Its definition appears in Figure 3. There are two changes with respect to *clwp*: (1) the occurrence of the persistently modality $\square$, and (2) the use of the postcondition $\_.\ \mathit{True}$ instead of a universally quantified one. The persistently modality is necessary to allow `call/cc`-captured continuations to be used arbitrarily: the assumption $\square \forall v,\ Q\ v \mathrel{-\!\!*} \mathit{wp}\ K[v]\ \{\, \_.\ \mathit{True} \,\}$ is used to justify that the reified context K in `kont` K can be restored multiple times via `throw`. The use of $\_.\ \mathit{True}$ is related to the statement of soundness (§6.5); see Footnote 4.

The *wp/cc* construction can be used in place of the protocol-based approach of Maze to deliver the logic from Figure 1. It can also be used, with further adjustments, to deliver the logic from Figure 2. The key difference between these logics is the support for the frame rule: the logic from Figure 2 supports it whereas the logic from Figure 1 does not. If (like *clwp*) the definition of *wp/cc* closes under the bind rule, then now we apply the same technique to close our initial *wp/cc* construction under the frame rule. This construction is written $\mathit{wp/cc}\ e\ \{\!\{\, Q \,\}\!\}$. Its definition appears in Figure 3. It closes $\mathit{wp/cc}\ e\ \{\, Q \,\}$ under the frame rule by stating that, if $\mathit{wp/cc}\ e\ \{\!\{\, Q \,\}\!\}$ hods, then, for every resource R, if R is available now, then R is available when e terminates.

The triples from Figures 1 and 2 are defined on top of *wp/cc* in the usual way as shown in Figure 3. We also introduce *one-shot triples* $\{\, P \,\}\ e\ \{\, Q \,\}_1$ and $\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}_1$, which follow the same construction as usual triples but avoid the persistently modality. One-shot triples are well-suited for reasoning about one-shot continuations. They are used, for example, in the definition of $\mathit{isCont}_1\ k\ Q$ (Figure 3) to formalize the intuition that Q is the precondition of k. The definition of *isCont* follows the same structure, but uses a standard triple instead of a one-shot triple. Because the persistently modality can always be eliminated, standard triples can be weakened to one-shot triples and, in particular, a multi-shot continuation can be used as a one-shot continuation:

$$\mathit{isCont}\ k\ Q \vdash \mathit{isCont}_1\ k\ Q$$

6.3 Deriving the rules of the logics

Some of the rules from Figures 1 and 2, such as the bind rule, the frame rule, and the `throw` rules, follow directly from the logical definitions of Figure 3. Other rules are derived by unfolding *wp/cc* in terms of *wp* and by applying the operational-style rules of *wp* we briefly discussed in §6.2. For the sake of illustration, let us explain the derivation of Rule (Call/1cc0).

After unfolding *wp/cc*, introducing the precondition P, an abstract resource R, and an abstract context K for which $\square\,\forall v,\ (Q\ v * R) \mathrel{-\!\!*} wp\ K[v]\ \{\ \_.\ True\ \}$ holds, the goal becomes:

$$wp\ K[\texttt{call/1cc0}\ (\lambda \texttt{k}.\ e)]\ \{\ \_.\ True\ \}$$

Rule (WpCall/1cc0) then further reduces the goal to the following assertion

$$\forall \ell,\ \ell \mapsto () \mathrel{-\!\!*} wp\ (e[\texttt{k} := \texttt{cont}\ \ell\ K]\texttt{;}\ \texttt{assert false})\ \{\ \_.\ True\ \}$$

After introducing the location ℓ and the points-to assertion $\ell \mapsto ()$, we apply e's specification $\forall k,\ \{\!\{\ P * isCont_1\ k\ Q\ \}\!\}\ e[\texttt{k} := k]\ \{\!\{\ \_.\ False\ \}\!\}$ with k instantiated as $\texttt{cont}\ \ell\ K$ and with the underlying universally quantified context in the definition of *wp/cc* instantiated with $\bullet\texttt{;}\ \texttt{assert false}$. The premises of e's specification are easily satisfied with the exception of $isCont_1\ (\texttt{cont}\ \ell\ K)\ Q$. By repeatedly unfolding definitions, the proof of $isCont_1\ (\texttt{cont}\ \ell\ K)\ Q$ reduces to the following goal where v is a value satisfying Q and K' is an arbitrary context:

$$wp\ K'[\texttt{throw}\ (\texttt{cont}\ \ell\ K)\ v]\ \{\ \_.\ True\ \}$$

After consuming $\ell \mapsto ()$, Rule (WpCont) further advances the proof to:

$$wp\ K[v]\ \{\ \_.\ True\ \}$$

To conclude, it suffices to apply $\square\,\forall v,\ (Q\ v * R) \mathrel{-\!\!*} wp\ K[v]\ \{\ \_.\ True\ \}$ with $Q\ v$ and R.

6.4 Connecting the logics

Because *wp/cc* is a shared foundation between the logics of Figures 1 and 2, it is possible to connect these logics thus enabling the verification of programs that mix `call/cc` and `call/1cc`:

$$\textsc{WithFrame}\ \frac{\{\!\{\ P\ \}\!\}\ e\ \{\!\{\ Q\ \}\!\}}{\{\ P\ \}\ e\ \{\ Q\ \}} \qquad \textsc{NoFrame}\ \frac{\forall R,\ \{\ P * R\ \}\ e\ \{\ v.\ Q\ v * R\ \}}{\{\!\{\ P\ \}\!\}\ e\ \{\!\{\ Q\ \}\!\}}$$

Rule (WithFrame) shows that it is possible to switch from the `call/cc` logic of Figure 1 to the `call/1cc` logic of Figure 2, thereby losing reasoning support for `call/cc`, but gaining support for the frame rule and for `call/1cc`. Its derivation follows by instantiating the abstract resource R in the definition of $wp/cc\ \_\ \{\!\{\ \_\ \}\!\}$ with *True*.

Rule (NoFrame) shows that a program fragment e that uses `call/cc` can still be integrated into a larger program with no damage to the frame rule, provided that e is shown to preserve any *residual heap* R. Its derivation follows directly from the definition of $wp/cc\ \_\ \{\!\{\ \_\ \}\!\}$.

6.5 Soundness

As usual in Iris-based logics, soundness is shown via *adequacy*:[4]

Theorem 6.1 (Adequacy of *wp/cc*). *If* $wp/cc\ e\ \{\ \_.\ True\ \}$ *or* $wp/cc\ e\ \{\!\{\ \_.\ True\ \}\!\}$, *then* e *is* safe.

This theorem shows that, if a specification for e can be proved in the logic, then this program must be *safe* with respect to the operational rules from §6.1; in particular, e either diverges or terminates. The proof is simple. It relies on the fact that $wp/cc\ e\ \{\ \_.\ True\ \}$ (and so $wp/cc\ e\ \{\!\{\ \_.\ True\ \}\!\}$) implies $wp\ e\ \{\ \_.\ True\ \}$ and that Iris's *wp* is adequate (a theorem proven in Iris once and for all):

Theorem 6.2 (Adequacy of *wp*). *If* $wp\ e\ \{\ \_.\ True\ \}$, *then* e *is safe.*

[4]Typically, the statement of adequacy allows a *pure* postcondition ϕ instead of $\_.\ True$, and guarantees that, if e terminates with a value v, then $\phi\ v$ holds in the meta logic. The limitation to $\_.\ True$ can be traced back to its occurrence in the definition of *wp/cc*. Another option is to parameterize the definitions from Figure 3 over a *global* postcondition ϕ introduced during the proof of adequacy and to use ϕ instead of $\_.\ True$ in the definition of *wp/cc*. In this manner, the more general adequacy statement could be proven. However, we use $\_.\ True$ for simplicity and because it is sufficient for soundness.

```
let with_fork main = call/1cc0 (λreturn.
  let q = Queue.create () in
  let next () =
    if Queue.empty q then throw return () else
      let k = Queue.take q in throw k ()
  in
  let fork task = call/1cc0 (λk. Queue.add k q; task (); next ()) in
  let yield () = call/1cc0 (λk. Queue.add k q; next ()) in
  main fork yield;
  next ()
)
```

Fig. 4. Cooperative-multithreading library.

6.6 Support for concurrency

The language has support for sequentially consistent concurrency via a `fork` instruction and atomic memory operations (such as `CAS` and `FAA`). The logic has support for concurrency via rules for opening and closing *invariants* using Iris's mechanism of *masks*. None of the case studies directly use concurrent instructions. Moreover, only one of them uses invariants (§7.3), but we do not discuss this case study here in detail. Therefore, for the sake of conciseness and simplicity, we refer the reader to the Rocq formalization [Anonymous authors 2026] for the formal semantic account of concurrency, the rules for invariants, and the definition of *wp/cc* with masks.

7 Applications of the logic

We exercise our logic for `call/1cc0` in a number of interesting case studies. For readability, we write these case studies in an OCaml-like syntax, which we manually translate to our formalized language in order to carry out their verification in Rocq.

7.1 Cooperative multithreading

One of the main motivations for introducing programming support for continuations is that, with continuations, multithreading constructs can be implemented as custom, user-defined library functionalities, rather than built-in primitives. It is indeed the primary motivation behind the adoption of effect handlers in OCaml [Leroy et al. 2026, Chapter 12]. Here we show that `call/1cc0` can be used to implement a multithreading library despite its underlying one-shot restriction. We also discuss how to verify this implementation using our logic.

7.1.1 Implementation. Figure 4 shows the implementation of our multithreading library. The library is presented as a function `with_fork` that supplies the client `main` with a fork functionality, to spawn new threads, and a yield functionality, to voluntarily release control to another thread. Internally, threads are represented as continuations. Except for the running thread, all threads are suspended and stored in the queue `q`. Forking a thread to run `task` has the effect of suspending a thread's own execution and running `task`. This is achieved in the implementation of `fork` via `call/1cc0`, which captures the running thread's continuation, stores it in `q`, then runs `task`. The implementation of `yield` is similar. Both terminate with a call to `next`, which either resumes one of the suspended threads from the queue, if the queue is nonempty, or invokes the continuation `return`.

Specification of `fork` **and** `yield`**.**

$$\begin{aligned} \mathit{isFork}\ R\ \mathit{fork} &= \square \forall \mathit{task},\ \{\!\{ R \}\!\}\ \mathit{task}\ ()\ \{\!\{ \_.\, R \}\!\} \mathrel{-\!\!*} \{\!\{ R \}\!\}\ \mathit{fork}\ \mathit{task}\ \{\!\{ \_.\, R \}\!\} \\ \mathit{isYield}\ R\ \mathit{yield} &= \{\!\{ R \}\!\}\ \mathit{yield}\ ()\ \{\!\{ \_.\, R \}\!\} \end{aligned}$$

Specification of `with_fork`**.**

$$\frac{\forall R\ \mathit{fork}\ \mathit{yield},\ \{\!\{ R * \mathit{isFork}\ R\ \mathit{fork} * \mathit{isYield}\ R\ \mathit{yield} \}\!\}\ \mathit{main}\ \mathit{fork}\ \mathit{yield}\ \{\!\{ \_.\, R \}\!\}}{\{\!\{ \mathit{True} \}\!\}\ \texttt{with\_fork}\ \mathit{main}\ \{\!\{ \_.\, \mathit{True} \}\!\}}$$

Fig. 5. Specification of the cooperative-multithreading library.

This continuation is obtained at the beginning of `with_fork`'s execution. It allows `with_fork` to terminate by returning to the original position where `with_fork` was called.

7.1.2 Specification and verification. The specification of `with_fork` appears in Figure 5. Because the function `with_fork` itself does not compute anything meaningful, its postcondition is $\_.\ \mathit{True}$. This guarantees that `with_fork` is memory safe: using `with_fork` does not lead to memory errors. Moreover, the specification guarantees that the functions *yield* and *fork* supplied to the client *main* are correct. This is expressed in *main*'s specification, which appears as a premise of `with_fork`'s specification.

The predicates *isYield* and *isFork* respectively capture the correctness of *yield* and *fork*. Both depend on an exclusive resource R that works as the permission to call these functions. The assertion $\mathit{isYield}\ R\ \mathit{yield}$ states that, to yield control, ownership of R must also be transferred to the next running thread and that, upon resumption, ownership of R is regained. The assertion $\mathit{isFork}\ R\ \mathit{fork}$ expresses the same ownership transferring mechanism, but, on top of that, also allows the new thread *task* to yield control and fork threads thanks to the permission R in its precondition.

The key idea in the verification of `with_fork` is to instantiate *main*'s specification with a suitable definition of the permission R. Intuitively, the ownership of this permission should reflect ownership of the ephemeral structures manipulated by `with_fork` and describe invariants over these structures. This is precisely how we define R:

$$R\ =\ \mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True}) * \exists \mathit{ks},\ \mathit{isQueue}\ q\ \mathit{ks} * \left(\textstyle\ast_{k \in \mathit{ks}} \triangleright\, \mathit{isCont}_1\ k\ (\lambda\_.\ R)\right)$$

It is a recursive definition that relies on Iris's *guarded recursion fixpoints*[5]. It states that *return* is a valid one-shot continuation and that q is a queue storing a set of suspended threads ks[6], each of which is ready to be resumed provided the permission R itself is supplied. The continuation *return* is the value to which `return` is bound in the beginning of `with_fork`'s execution. Because the postcondition of `with_fork` is $\_.\ \mathit{True}$, it is easy to see that, after the application of (Call/1cc0), the assertion $\mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True})$ holds. Moreover, because the postconditions of *yield* and *fork* are both $\_.\ R$, it follows analogously that, after applying (Call/1cc0), the captured continuation k satisfies $\mathit{isCont}_1\ k\ (\lambda\_.\ R)$ as required to maintain the queue invariant. With the definition of R and with these observations in mind, the verification of `with_fork` follows naturally.

7.2 Control inversion

Implementations of data structures often provide *iterators* to perform an action over all elements of a given structure. Functional languages favor *internal iterators*, which are higher-order functions

[5]In short, this means that recursive definitions are allowed as long as the recursive occurrences are guarded by the later modality $\triangleright$, which can be eliminated during the proof whenever the program advances by one reduction step.
[6]The predicate *isQueue* comes as part of a verified queue library.

```
let invert iter = λ(). call/1cc0 (λkc.
  let r = ref kc in
  let yield x = call/1cc0 (λkp.
    throw !r (Seq.Cons (x, λ(). call/1cc0 (λkc. r := kc; throw kp ()))))
  in
  iter yield; throw !r Seq.Nil
)
```

Fig. 6. Implementation of control inversion using `call/1cc0`.

that apply a user-provided function to each element. An example is OCaml's `List.iter` function, with type `('a -> unit) -> 'a list -> unit`. Object-oriented languages favor *external iterators*, which are objects or functions that can enumerate the elements of a given structure, returning the next element each time the iterator is called by user code. In a functional language, external iterators can be represented using a *lazy sequence*, a suspension that, when forced, produces either a pair `Seq.Cons` (x, s) of the next element x and a new sequence s denoting the remaining elements or the value `Seq.Nil` denoting an empty sequence.

Continuations can be used to convert an internal iterator to an external iterator. This is an instance of *control inversion*. Implementations of control inversion using effect handlers and `call/cc` have been studied in previous work [de Vilhena 2022, 2026]. Here, we verify a novel implementation using `call/1cc0`.

7.2.1 Implementation. The implementation appears in Figure 6.[7] As usual with applications of `call/1cc0` (and of undelimited continuations in general), it is helpful to think in terms of two communicating agents: in this case, a *consumer* and a *producer*. The producer is the internal iterator: it is the code that yields elements. The consumer is `invert`'s client: it is the code that requests elements by forcing the lazy sequence returned by `invert`.

The main idea in this implementation is to enable control to switch back and forth between consumer and producer. The implementation achieves this by using `call/1cc0` to obtain and store the *position* (that is, the program point) of either the consumer or the producer before each jump. The reference `r` stores the position of the consumer, whereas the position of the producer is implicitly stored (as a closure-captured value) in each lazy sequence sent from the producer to the consumer.

When the consumer requests the first element, `invert` uses the initial `call/1cc0` to obtain the consumer's position and to store it in `r`. `invert` then proceeds with the execution of `iter yield`. When `iter` stumbles upon an element `x` and calls `yield x`, control is transferred back to the consumer by invoking the continuation stored in `r` with `x` and a lazy sequence for the remaining elements. However, before this jump, `yield` uses `call/1cc0` to obtain its own position `kp`. When the consumer requests a further element, it uses `call/1cc0` to update `r` with its new position and it uses `kp` to jump back to the producer and resume `iter`. Eventually, when `iter` terminates, the producer jumps back to the consumer with `Seq.Nil`.

7.2.2 Specification and verification. `invert`'s implementation relies on a subtle control-switching mechanism. It is hard to be convinced of its correctness by studying this implementation alone. Thankfully, its verification using our logic is fairly straightforward and, by virtue of the frame and

[7] Here, the internal iterator `iter` is partially applied to a structure: it has type `('a -> unit) -> unit`. This is simply a stylistic choice; it is trivial to add the structure as an extra argument of both `iter` and `invert`.

Internal iterators.

$$\begin{aligned} &\mathit{isIter}\ \mathit{iter}\ \mathit{xs}\ =\ \square \forall I\ f,\\ &\quad (\forall \mathit{ys}\ z\ \mathit{zs}, \{\!\{\ I\ \mathit{ys}\ (z :: \mathit{zs})\ \}\!\}\ f\ z\ \{\!\{\ \_.\ I\ (\mathit{ys} +\!\!+ [z])\ \mathit{zs}\ \}\!\}) \mathrel{-\!\!*} \{\!\{\ I\ []\ \mathit{xs}\ \}\!\}\ \mathit{iter}\ f\ \{\!\{\ \_.\ I\ \mathit{xs}\ []\ \}\!\} \end{aligned}$$

Lazy sequences.

$$\begin{aligned} \mathit{isSeq}\ s\ \mathit{zs}\ &=\ \{\!\{\ \mathit{True}\ \}\!\}\ s\ ()\ \{\!\{\ h.\ \mathit{isHead}\ h\ \mathit{zs}\ \}\!\}_1\\ \mathit{isHead}\ h\ \mathit{zs}\ &=\ \texttt{match}\ h\ \texttt{with}\\ &\quad \texttt{| Seq.Nil ->}\ \mathit{zs} = []\\ &\quad \texttt{| Seq.Cons}\ (z,\ s')\ \texttt{->}\ \exists \mathit{zs}',\ \mathit{zs} = z :: \mathit{zs}' \ast \triangleright\ \mathit{isSeq}\ s'\ \mathit{zs}' \end{aligned}$$

Specification of `invert`.

$$\forall \mathit{iter}\ \mathit{xs},\ \{\!\{\ \mathit{isIter}\ \mathit{iter}\ \mathit{xs}\ \}\!\}\ \texttt{invert}\ \mathit{iter}\ \{\!\{\ s.\ \mathit{isSeq}\ s\ \mathit{xs}\ \}\!\}$$

Fig. 7. Specification of `invert`.

bind rules, shows that one can reason about `invert` locally regardless of the complex change of control flow in its operation. Once its specification is written out, the verification follows almost directly from `call/1cc0`'s reasoning rule.

The specification of `invert` appears in Figure 7. It follows closely the specification of de Vilhena [2026]'s `call/cc`-based implementation. The main difference is that, because `call/cc` captures persistent continuations, lazy sequences in de Vilhena [2026]'s implementation are also persistent whereas here they are ephemeral (as reflected by the use of one-shot triples).

The specification of `invert` is succinct: given an iterator *iter* for the elements *xs*, `invert` produces a lazy sequence *s* for the same set of elements. The expected behavior of the iterator is captured in the logic via the assertion *isIter iter xs*. This assertion says that, for any loop invariant I (indexed on the set of elements already seen and on the set of elements to be seen) and for any function f capable of advancing I by one element, *iter* f processes all elements advancing I from $I\ []\ \mathit{xs}$ to $I\ \mathit{xs}\ []$. The guaranteed behavior of a lazy sequence is captured in the logic via the assertion *isSeq s xs*, which states *s* is an ephemeral suspension that, when forced, produces a value *h* that is either `Seq.Nil` or a pair `Seq.Cons` $(z,\ s')$. The former case happens when the list is empty, whereas the latter case happens when z is the head element of *xs* and s' is a sequence representing the remaining elements.

Because `call/1cc0`'s rule does not introduce metavariables, verifying an application of `call/1cc0` is straightforward: the only option is to apply (Call/1cc0) and proceed with the proof. The crux in the verification of `invert` is reasoning about the *iter* step, because it requires using the *isIter iter xs* assumption which does introduce a metavariable: the loop invariant I. Fortunately, a simple definition suffices. It states that r (the location to which `r` is bound at runtime) stores a continuation *kc* that can only be resumed with a value *h* such that *isHead h zs* (where *zs* are the remaining elements to be seen):

$$I\ \_\ \mathit{zs}\ =\ \exists \mathit{kc},\ r \mapsto \mathit{kc} \ast \mathit{isCont}_1\ \mathit{kc}\ (\lambda h.\ \mathit{isHead}\ h\ \mathit{zs})$$

7.3 Call/cc-one-shot

Friedman and Haynes [1985] introduce a `call/cc-one-shot` operator that wraps the continuation captured by `call/cc` with a one-shot check:

```
let call/cc-one-shot f = call/cc (λk,
  let b = ref false in f (λy, (if !b then error); b := true; throw k y))
```

```
let hs = Stack.create ()

let abort res = let k = Stack.pop hs in throw k res

let perform x = call/1cc0 (λk, abort (Effect (x, k)))

let rec handle { effc; retc } main =
  match call/1cc0 (λk.
    Stack.push k hs; let y = main () in abort (Return y))
  with
  | Return y ->
      retc y
  | Effect (x, k) ->
      effc x (λy. handle { effc; retc } (λ(). throw k y))
```

Fig. 8. Implementation of effect handlers using `call/1cc0`.

This construct differs from `call/1cc` in that the context in which `f` runs is not empty: it is the same as the reified context in `k`. Therefore, technically speaking, copies of the continuation can still be created by re-executing `call/cc-one-shot` under the same context. Although the difference is subtle, this allows `call/cc-one-shot` to implement `call/cc`, as shown by Friedman and Haynes:

```
let call/cc* f =
  let r = ref (λ_, assert false) in
  let rec G f =
    let x = call/cc-one-shot (λk, r <- (throw k); f (λy, !r y)) in
    call/cc-one-shot (λk, G (λ_, throw k x))
  in G f
```

This implementation of `call/cc` on top of `call/cc-one-shot` attracted the attention of many authors, who have shown its correctness using a variety of techniques [Dreyer et al. 2010; Støvring and Lassen 2007]. Timany and Birkedal [2019], for example, show contextual equivalence between `call/cc*` and `call/cc`. Our logic offers no way to compare programs, but we can show that the implementation of `call/cc*` admits the same reasoning rule as `call/cc`, namely (Call/cc). The verification is technical and offers little insight about one-shot undelimited continuations. In the interest of space, we refer the reader to our Rocq formalization for more details [Anonymous authors 2026].

8 Effect handlers and undelimited continuations

We conclude the technical discussion of the paper with an exploration of the relation between effect handlers and undelimited continuations through the perspective of our logic.

8.1 Continuation-based implementation of effect handlers

The starting point of our exploration is Filinski [1994, 1996]'s celebrated result showing that `shift/reset` can be implemented on top of `call/cc`, or, in other words, that delimited control operators can be expressed in terms of undelimited continuations. Inspired by Filinski's encoding,

we derive a similar result for one-shot undelimited continuations: namely, that effect handlers can be implemented via `call/1cc0`. We then show that, using our logic, we can verify this implementation with respect to a set of abstract specifications using de Vilhena and Pottier [2021]'s protocols. Finally, we discuss how this difference in abstraction layers justifies Kiselyov [2012a]'s criticisms.

8.1.1 Implementation. Our `call/1cc0`-based implementation of effect handlers appears in Figure 8. The operation `perform` offers the ability to perform an effect with payload `x`. The operation `handle` installs a handler with effect branch `effc` and return branch `retc` to handle effects of the computation `main`. This computation is represented as a thunk (because our language is strict). The stack `hs` and the operation `abort` are used internally in the implementation of `handle` and `perform`, but not exposed otherwise.

In the usual dynamic semantics of handlers, an effect handler is represented as a frame in the control stack; when an effect is performed, control is transferred to the nearest such handler. The key idea in the implementation of Figure 8 is to mimic this behavior with `hs`, a heap-allocated stack of continuations, each representing a handler. The continuation at the top of `hs` is the nearest handler.

The function `handle` obtains its continuation with `call/1cc0`, pushes it on the stack, and proceeds with the execution of `main`. One of two outcomes may follow: either `main` terminates normally with a value `y` or it performs an effect with payload `x`. In both cases, `abort` jumps to the nearest enclosing handler `k`, which it pops from `hs`. In case of normal termination, `abort` jumps to the handler with `Return y` and the `match` expression in `handle` then reduces to the return branch `retc`. In case of an effect with payload `x`, `abort` jumps to the handler with `Effect (x, k)`, where `k`, the continuation captured by the `call/1cc0` in `perform`, records the position where the effect was performed and allows the handler to resume the computation. In this case, the `match` expression in `handle` reduces to `effc`. The continuation supplied to the effect branch `effc` however is not merely `k` (the continuation captured by `perform`): instead, the supplied continuation wraps `k` in a recursive application of `handler`. This simulates the semantics of deep handlers, which are reinstalled when resuming a continuation.

Comparison with Filinski's encoding. As mentioned previously, our implementation closely follows Filinski [1994, 1996]'s encoding of `shift` and `reset`:[8]

```
let mk = ref (λ_. assert false)
let abort y = !mk y
let reset comp = call/cc (λk.
  let m = !mk in mk := (λy. mk := m; throw k y); abort (comp ()))
let shift f = call/cc (λk. abort (f (λy. reset (λ(). throw k y))))
```

The reference `mk` plays a similar role as the stack `hs` in Figure 8. If `hs` simulates a control stack of handlers, then `mk` simulates a control stack of `reset`s. However, instead of storing each continuation `k` captured by `reset` in an explicit heap-allocated stack, `mk` leverages the fact that it stores a function to simulate push and pop operations in the function itself. Indeed, `reset` updates `mk` with a closure capturing `m`, the previous state of `mk`, and `k`, the continuation captured via `call/cc` by `reset`. When applied to a value `y` (during the execution of `abort`) this closure will restore `m` in `mk` before invoking `k` with `y`.

8.1.2 Specification and verification. The specification of `perform` and `handle` appears in Figure 9. The specification is expressed in a logical abstraction layer built on top of *wp/cc*. The key ingredient

[8]Here, we adapt Filinski's encoding (originally written in Standard ML) to our OCaml-like syntax.

Abstraction layer.

$$\begin{aligned}
\mathit{wp/hs}\ e\ \langle\, H \,\rangle\ \{\, Q \,\} &= \mathit{canAbort}\ H \mathrel{-\!\!*} \mathit{wp/cc}\ e\ \{\!\!\{\, y.\, Q\ y * \mathit{canAbort}\ H \,\}\!\!\} \\
\{\!\!\{\, P \,\}\!\!\}\ e\ \langle\, H \,\rangle\ \{\!\!\{\, Q \,\}\!\!\}_1 &= P \mathrel{-\!\!*} \mathit{wp/hs}\ e\ \langle\, H \,\rangle\ \{\, Q \,\} \\
\{\!\!\{\, P \,\}\!\!\}\ e\ \langle\, H \,\rangle\ \{\!\!\{\, Q \,\}\!\!\} &= \Box \{\!\!\{\, P \,\}\!\!\}\ e\ \langle\, H \,\rangle\ \{\!\!\{\, Q \,\}\!\!\}_1
\end{aligned}$$

Internal logical definitions.

$$\begin{aligned}
\mathit{canAbort}\ H &= \exists \mathit{ks},\ \mathit{isStack}\ \mathit{hs}\ \mathit{ks} * \mathit{isHStack}\ \mathit{ks}\ H \\
\mathit{isHStack}\ \mathit{ks}\ H &= \texttt{match}\ (\mathit{ks}, H)\ \texttt{with} \\
&\quad \mid (k :: \mathit{ks}', (\Psi, Q) :: H')\ \texttt{->}\ \mathit{isHStack}\ \mathit{ks}'\ H' * \\
&\qquad \mathit{isCont}_1\ k\ (\lambda \mathit{res}.\ \mathit{mainPost}\ \Psi\ Q\ H'\ \mathit{res} * \mathit{canAbort}\ H') \\
&\quad \mid ([], [])\ \texttt{->}\ \mathit{True} \\
&\quad \mid _\ \texttt{->}\ \mathit{False} \\
\mathit{mainPost}\ \Psi\ Q\ H\ \mathit{res} &= \texttt{match}\ \mathit{res}\ \texttt{with} \\
&\quad \mid \texttt{Return}\ y\ \texttt{->}\ Q\ y \\
&\quad \mid \texttt{Effect}\ (x, k)\ \texttt{->}\ \exists Q',\ \Psi\ x\ Q' * \triangleright\, \mathit{isCont}_1\ k\ (\lambda y.\, Q' y * \mathit{canAbort}\ H))
\end{aligned}$$

Specification of `perform` **and** `handle`.

PEFORM

$$\frac{}{\{\!\!\{\, \Psi\ x\ Q' \,\}\!\!\}\ \texttt{perform}\ x\ \langle\, (\Psi, Q) :: H \,\rangle\ \{\!\!\{\, Q' \,\}\!\!\}}$$

HANDLE

$$\frac{\{\!\!\{\, P \,\}\!\!\}\ \mathit{main}\ ()\ \langle\, (\Psi, Q) :: H \,\rangle\ \{\!\!\{\, Q \,\}\!\!\}}{\{\!\!\{\, P * \mathit{isHandler}\ \Psi\ Q\ \mathit{effc}\ \mathit{retc}\ H\ Q' \,\}\!\!\}\ \texttt{handle}\ \{\ \mathit{effc}\ ;\ \mathit{retc}\ \}\ \mathit{main}\ \langle\, H \,\rangle\ \{\!\!\{\, Q' \,\}\!\!\}}$$

Fig. 9. Specification of `call/1cc0`-based implementation of handlers.

in the specification as well as in all internal definitions that appear in Figure 9 are *protocols* [de Vilhena and Pottier 2021], so let us start with a brief explanation of this notion.

Protocol primer. A protocol is a succinct way to provide the specification of a function. In this sense, the notion of a protocol is independent from effects. However, protocols are useful in the context of effects and handlers precisely because they provide a way to specify the functionality implemented by a handler.

Formally, a protocol Ψ is a predicate transformer that, given a value x and a postcondition Q, computes a precondition $\Psi\ x\ Q$. A function f is correct with respect to a protocol Ψ if the triple $\forall x\ Q,\ \{\!\!\{\, \Psi\ x\ Q \,\}\!\!\}\ f\ x\ \{\!\!\{\, Q \,\}\!\!\}$ holds.[9] Often, Ψ has shape $\lambda x\ \Psi.\, P\ x\ *\ (\forall y, Q\ x\ y \mathrel{-\!\!*} Q\ y)$. In this case, the specification provided by Ψ essentially unfolds to Iris's *Texan triple*, $\forall x\ Q,\ \{\!\!\{\, P\ x * \forall y, Q\ x\ y \mathrel{-\!\!*} Q\ y \,\}\!\!\}\ f\ x\ \{\!\!\{\, Q \,\}\!\!\}$, whereby, in addition to $P\ x$, the precondition includes $\forall y, Q\ x\ y \mathrel{-\!\!*} Q\ y$, the permission to terminate. Such a Texan triple can be read as a standard triple $\forall x,\ \{\!\!\{\, P\ x \,\}\!\!\}\ f\ x\ \{\!\!\{\, y.\, Q\ x\ y \,\}\!\!\}$, but, by virtue of quantifying universally over the postcondition, it is easier to use and more amenable to automation than a standard triple. We use protocols, and therefore this style of triples, because of their combination of generality and succinctness.

[9] We write the triple in our logic, but this construction works with any other notion of triple.

Specification of `perform` *and* `handle`. The interface between a handler and an effectful computation is specified by a pair (Ψ, Q), where Ψ specifies the effectful functionality implemented by the handler and Q specifies the value returned by the computation. The *weakest precondition with a handler stack*, written *wp/hs* $e \langle H \rangle \{ Q \}$, includes a list H of such pairs, each of which representing a handler on the stack. On top of *wp/hs* we derive triples parameterized by H.[10]

The specification of `perform` assumes the innermost handler implements an effect specified by Ψ, therefore, to perform an effect with payload x expecting in return a result satisfying Q', the precondition $\Psi\ x\ Q'$ must hold. This is in line with the interpretation of a protocol Ψ as a predicate transformer that computes a precondition.

The specification of `handle` extends the handler stack H with a pair (Ψ, Q) provided that the effect branch *effc* and the return branch *retc* are correct with respect to Ψ and Q. This is expressed via the *handler judgment*:

$$
\begin{array}{l}
\mathit{isHandler}\ \Psi\ Q\ \mathit{effc}\ \mathit{retc}\ H\ Q' \ = \\
\quad \left(\forall y,\ \{\!\{ Q\ y \}\!\}\ \mathit{retc}\ y \langle H \rangle \{\!\{ Q' \}\!\}_1\right) \ \wedge \\
\quad \left(\begin{array}{l}
\forall x\ k\ Q'', \\
\{\!\{ \Psi\ x\ Q'' * (\forall y\ Q',\ \{\!\{ Q''\ y * \triangleright\ \mathit{isHandler}\ \Psi\ Q\ \mathit{effc}\ \mathit{retc}\ H\ Q' \}\!\}\ k\ y \langle H \rangle \{\!\{ Q' \}\!\}_1) \}\!\} \\
\quad \mathit{effc}\ x\ k \\
\langle H \rangle \{\!\{ Q' \}\!\}_1
\end{array}\right)
\end{array}
$$

isHandler $\Psi\ Q\ \mathit{effc}\ \mathit{retc}\ H\ Q'$ is defined as the non-separating conjunction of two assertions: the specification of *retc* and the specification of *effc*. The specification of *retc* says that, given a value y such that $Q\ y$ holds, *retc* returns a value according to `handle`'s postcondition Q'. The specification of *effc* includes $\Psi\ x\ Q''$ and a specification of k as preconditions. The value x is the payload with which the computation performed the effect, k is the `perform`-captured continuation wrapped in a recursive application of the handler, and Q'' is the postcondition of `perform` expected by the computation. From the perspective of the computation, the assertion $\Psi\ x\ Q''$ is an obligation to perform the effect, but, from the perspective of the effect branch, it appears as an assumption. Similarly, from perspective of the computation, Q'' is the postcondition of `perform`, but, from the perspective of *effc*, it appears as the precondition to resume k. Because the continuation k is one-shot, it is specified by a one-shot triple. Moreover, because it is wrapped in a recursive application of the handler, it further requires *isHandler*. The use of one-shot triples to specify *retc* and *effc* strengthens the overall specification of `handle` by allowing resources to be transferred to *retc* and *effc*.

Verification. We carry out the verification of `perform` and `handle` at the level of our *wp/cc*-based logic, after unfolding the definition of *wp/hs*. This exposes the assertion *canAbort* H, which is assumed in the precondition and required in the postcondition. This assertion is the permission to call `abort`. Its definition (Figure 9) claims ownership over *hs* (the runtime value to which `hs` is bound) and formalizes the assumption that *hs* stores a stack of handlers specified by the list of protocol-predicate pairs H. It does so by asserting the existence of a list of continuations *ks* stored in *hs*[11] and by asserting *isHStack ks H*. This is the assertion that links the `handle`-captured continuations *ks* to their logical specification H. It is defined inductively. The base case, when both *ks* and H are [], holds trivially. The case where *ks* is a nonempty list $k :: ks'$ and H is $(\Psi, Q) :: H'$ holds (1) if ks' is specified by H', that is, if *isHStack* $ks'\ H'$ holds, and (2) if k is a continuation that, to be resumed, requires the permission *canAbort* H' and a value *res* such that *mainPost* $\Psi\ Q\ H'\ \mathit{res}$. The assertion *mainPost* $\Psi\ Q\ H\ \mathit{res}$, as the name suggests, restricts *res* to one of the two outcomes of *main*:

[10] Technically, $\{\!\{ P \}\!\}\ e \langle H \rangle \{\!\{ Q \}\!\}$ is a quadruple, but we use *triple* to mean *a specification with pre- and post- conditions.*

[11] This relation between *hs* and *ks* is expressed by the predicate *isStack*, which is included as part of a verified stack library.

```
let with_perform { effc; retc } main =
  let r = ref None in let abort res = thow (Option.get !r) res in
  let rec handle comp =
    match call/1cc0 (λk. r := Some k; let y = comp () in abort (Return y)) with
    | Return y -> retc y
    | Effect (x, k) -> effc x (λy. handle (λ(). throw k y))
  in
  let perform x = call/1cc0 (λk, abort (Effect (x, k))) in
  handle (λ(). main perform)
```

Fig. 10. Context-independet effects and handlers.

either a value `Return` y such that $Q\ y$, or a value `Effect` (x, k) such that (1) the precondition $\Psi\ x\ Q'$ required to perform an effect with payload x and to assign Q' as the postcondition of `perform` x holds, and (2) k is a continuation that expects the permission *canAbort* H and a value satisfying Q'.

With these definitions, we can prove the following specification for `abort`:

$$\forall \Psi\ Q\ H\ res,\ \{\!\{\ mainPost\ \Psi\ Q\ H\ res * canAbort\ ((\Psi, Q) :: H)\ \}\!\}\ \texttt{abort}\ res\ \{\!\{\ \_.\ False\ \}\!\}$$

The verification then follows mostly directly. The subtlest step is in the verification of `handle`, where, by exploiting the bind rule and the consequence rule, we focus on the `call/1cc0` expression and adjust its postcondition to $Q' = \lambda res.\ mainPost\ \Psi\ Q\ H\ res * canAbort\ H$ before applying (Call/1cc0), to obtain a continuation with precondition Q'. This is required to verify the push operation and, consequently, to update *canAbort* from H to $((\Psi, Q) :: H)$ (as required to run *main*).

Kiselyov's remarks. Kiselyov [2012a] discusses side conditions to Filinski [1994]'s result: in particular, the encoding of `shift`/`reset` using undelimited continuations assumes that the derived `shift` and `reset` constructs are the only forms of non-local control flow, thereby excluding exceptions and even `call/cc` itself. The same side conditions apply to our `call/1cc0`-based implementation of effect handlers: we assume `perform` is the only non-local control operator. Interestingly, this assumption is implicitly expressed by the abstraction layer in which the specification of `perform` and `handle` is written. Whereas *wp/cc* and *wp/hs* share most of the reasoning rules (for example, the bind and frame rules and the rules for state), only *wp/cc* offers rules for `call/cc`, `call/1cc0`, and undelimited continuations in general. This means that one cannot reason about programs that mix the `call/1cc0`-based implementation of `perform`/`handle` with `call/cc` or `call/1cc0` without breaking the abstraction layer of *wp/hs*. In other words, using `call/cc` or `call/1cc0` breaks the abstraction of performing and handling effects as provided by `perform` and `handle`.

8.2 Context-independent effects and handlers

Kiselyov's comments invite the question of whether it is possible to provide an interface for effect handlers that is compatible with undelimited continuations. We answer this question affirmatively with the function `with_perform` whose implementation appears in Figure 10. The function `with_perform` supplies a user-provided computation `main` with the functionality to perform effects, which are interpreted according to the (also user-provided) effect and return branches, `effc` and `retc`.

Implementation. The implementation of `with_perform` and of the internally defined `perform` operation that is supplied to `main` is very similar to the implementation of `handle` and `perform` from

$$\frac{\left(\begin{array}{l}\forall R\ \mathit{perform},\\ \{\!\{\ (\forall x\ Q,\ \{\!\{\ \Psi\ x\ Q * R\ \}\!\}\ \mathit{perform}\ x\ \{\!\{\ y.\, Q\ y * R\ \}\!\}) * R * P\ \}\!\}\ \mathit{main}\ \mathit{perform}\ \{\!\{\ y.\, Q\ y * R\ \}\!\}\end{array}\right)}{\{\!\{\ \mathit{isCIHandler}\ \Psi\ Q\ \mathit{effc}\ \mathit{retc}\ Q' * P\ \}\!\}\ \texttt{with\_perform}\ \{\ \mathit{effc};\ \mathit{retc}\ \}\ \mathit{main}\ \{\!\{\ Q'\ \}\!\}}$$

Fig. 11. Specification of `with_perform`.

Figure 8. The main difference is that, instead of a heap-allocated stack of handler positions `hs` manipulated by every invocation of `perform`/`handle`, each invocation of the function `with_perform` creates a fresh location `r` that essentially establishes a communicating channel between handler and handlee. Instead of a stack, the reference `r` stores the position of the single handler to which the handlee can jump when performing an effect. Therefore, whereas the implementation from Figure 8 suggests the abstraction of a stack of handlers, `with_perform`'s implementation (Figure 10) suggests a message-passing-inspired abstraction of pairs of a handler and a handlee.

Context-independent effects. By establishing this unique interaction channel between handler and handlee, `with_perform` provides an interface for effect handling that generalizes the traditional semantics of control-stack-based handlers. We say `with_perform` provides an interface for *context-independent* effects and handlers, because the `perform` functionality it implements is interpreted correctly regardless of the context. Specifically, as we discuss shortly, the interpretation of `perform` is valid during the *lifetime* of `main`, but it does not need to be used locally by `main`. This is because undelimited continuations are used in `with_perform` to their advantage: undelimited continuations can be invoked in any context, they do not depend on the presence of a handler or marker on the control stack.

This behavior has the consequence of allowing `main` to perform effects in combination with other forms of non-local control flow including `call/1cc`, nested applications of `with_perform`, and concurrency. For example, the following program (which uses `with_perform` to implement a counter in state-passing style) returns 2, showing both concurrent updates are correctly handled:

```
with_perform { effc = (λ() k s. k () (s + 1)); retc = (λ() s. s) } (λincr.
  let lock = Mutex.create () in
  let incr_with_lock () = Mutex.lock lock; incr (); Mutex.unlock lock in
  ( incr_with_lock () || incr_with_lock () )
) 0
```

As another illustrative example, the following program uses two nested `with_perform` applications to implement the effects `f` and `g`, and lets the outer handler, which handles `f`, perform `g`, which is then correctly handled by the inner `g` handler (as a function that takes and returns unit):

```
with_perform { effc = (λg k. g (); k ()); retc = (λ(). ()) } (λf.
  with_perform { effc = (λ() k. k ()); retc = (λ(). ()) } (λg. f g))
```

Specification. The formal specification of `with_perform` that appears in Figure 11 helps to formalize the programming interface provided by `with_perform`, its guarantees, and its conditions.

The specification states `with_perform` supplies `main` with a correct implementation of an operation *perform* provided that the user-supplied effect and return branches, *effc* and *retc*, are correct. Like in the specification of `handle` and `perform` (Figure 9), a protocol Ψ is used to specify the effectful functionality implemented by the handler and the predicate *isCIHandler* is used to capture the correctness assumptions on *effc* and *retc*. The assertion $\mathit{isCIHandler}\ \Psi\ Q\ \mathit{effc}\ \mathit{retc}\ Q'$ is defined in

the same way as *isHandler* $\Psi\ Q\ \mathit{effc}\ \mathit{retc}\ H\ Q'$, except that every occurrence of $\{\!\{ \_ \}\!\}\ \_\ \langle\, H \,\rangle\ \{\!\{ Q' \}\!\}_1$ is replaced with $\{\!\{ \_ \}\!\}\ \_\ \{\!\{ Q' \}\!\}_1$.

There are two main differences with respect to the specifications of `handle`/`perform` (Figure 9). First, instead of being expressed in a new abstraction layer, `with_perform`'s specification is entirely written using our *wp/cc*-based logic of Figure 2. This means, in particular, that (Call/1cc0) and `with_perform`'s specification can be used jointly to reason about programs with occurrences of both `call/1cc0` and `with_perform`.

Second, the pre- and post- conditions of *perform* x are not simply $\Psi\ x\ Q$ and Q (for universally quantified x and Q). Instead, both conditions include an exclusive resource R. This resource works as a *capability* to perform effects. It is granted to *main* in the beginning of its execution, required whenever *main* calls *perform*, and recovered when *perform* returns. It is by requiring R back in the postcondition of *main* that further calls to *perform* are disallowed when *main* terminates. In other words, calls to *perform* are allowed only during the execution of *main*. As previously discussed, this does not rule out calls to *perform* outside the scope of *main*: from a separation-logic point of view, these calls are justified as long as the ownership of R is correctly transferred.

Verification and applications. We verify that `with_perform` satisfies the specification from Figure 11. The key ideas are similar to the ones already discussed in §8.1.2, so we refer the reader to our formalization for more details [Anonymous authors 2026]. To ensure that `with_perform`'s proven specification is sufficiently expressive, we apply it to the verification of some simple applications, including an implementation of coroutines in the style of Python's generators:

```
let create g = ref (λ().
  with_perform { effc = (λx k. Seq.Cons (x, k)); retc = (λ(). Seq.Nil) } g)
let next c = match let Some g = !c in c := None; g () with
  | Seq.Cons (x, k) -> c := Some k; Some x
  | Seq.Nil -> None
```

9 Related work

9.1 Program logics for `call/cc`

Berger [2009] develops a program logic for PCF extended with `call/cc` and `throw`. The language does not support mutable state. The assertions of the logic allow one to name program points and associate logical invariants with them, in a way that is more complex than the *isCont* $k\ Q$ predicate used in the present paper.

Crolard and Polonowski [2012] study a functional and imperative language featuring Algol-style non-local jumps and mutable variables, but not general mutable references. The authors equip this language with a dependent type system that arises from a classical logic via the Curry-Howard correspondence, and then derive a Hoare logic from the type system.

Delbianco and Nanevski [2013] describe a separation logic for a functional and imperative language extended with `call/cc` and `abort`, which is a generalization of `throw` with better algebraic properties. This logic is presented in the style of Hoare Type Theory, using dependent types, and is formalized in Coq. The main limitation of this logic is that continuations cannot be stored in mutable references. Unfortunately, this rules out most practical uses of `call/cc`.

Timany and Birkedal [2019] build a separation logic for a HeapLang-style language of functions, general mutable references and concurrency, extended with `call/cc` and `throw`. This logic is built on top of Iris. It is the first program logic able to reason about stored continuations. The authors verified several realistic uses of `call/cc`, including an implementation of cooperative multithreading. This program logic is unusual in that it is defined for whole programs $K[e]$,

where K is a context and e is the subexpression under focus. The logic rules for `call/cc` and `throw` closely follow their operational semantics rules. This allows for precise reasoning about the successive invocations of a continuation. The downside of this approach is that compositional reasoning in general and the bind rule in particular are unsound. However, the authors provide sound context-local reasoning rules, including the bind rule, for subexpressions e that contain neither `call/cc` nor `throw`. As described in §6.2, our logic also builds on whole-program weakest preconditions, but in a different way that validates the bind rule.

De Vilhena [2026] develops a Hoare logic for a HeapLang-style language of functions and general mutable references extended with `call/cc` and `throw`. This logic is derived from the Maze logic of de Vilhena [2022, Chapter 6] via an encoding of `call/cc` and `throw` in terms of user-defined effects. As explained in §3, this logic supports the unrestricted bind rule, thus enabling modular reasoning on programs that use `call/cc`. However, it is not a full-fledged separation logic, since the frame rule only holds for expressions that do not use `call/cc` or `throw`.

9.2 Program logics for user-defined effects and effect handlers

De Vilhena and Pottier [2021] describe Hazel, a separation logic formalized in Iris for user-defined effects and effect handlers. As in OCaml 5, continuations captured by effect handlers are affine and can only be invoked once. The logic supports the bind rule and the frame rule without restrictions. A distinguishing feature of Hazel is its use of *protocols*, which are axiomatic specifications of the expected behavior of an effect and serve as the contract between users and handlers of effects. They enforce structured uses of effects and continuations. The work on `call/1cc` reported in this paper began with our desire to unravel the design of Hazel, separating the aspects related to the linearity of continuations from the aspects related to effect protocols.

The PhD thesis of de Vilhena [2022] describes Hazel in more detail, as well as Maze, a variant of Hazel that supports effect handlers with continuations that can be invoked multiple times. To maintain soundness, Maze restricts the frame rule to subexpressions that are verifiable in the empty protocol, *i.e.* those that do not perform effects.

Besides separation logics, other approaches have been proposed for reasoning about programs that use effects and handlers. The foundational work of Plotkin and Pretnar [2008, 2009] on denotational semantics for algebraic effects and handlers also includes an equational theory that proves equivalences between effectful programs. Matache and Staton [2019] define a logic to express properties of effectful programs. This logic is complete with respect to contextual equivalence.

9.3 One-shot continuations

The first paper to study the properties of one-shot continuations is Friedman and Haynes [1985]. However, the `call/cc-one-shot` control operator introduced in this paper differs from the `call/1cc` operator we study here: returning normally from `call/cc-one-shot` does not count as one use of the continuation. In other words, `call/cc-one-shot` implements *two-shot* continuations, which can be used twice, once by invoking the continuation and once by returning from `call/cc-one-shot`. This makes a major difference in expressiveness: Friedman and Haynes [1985] show that `call/cc-one-shot` plus mutable variables can implement the unrestricted `call/cc` operator, with multi-shot continuations. Thielecke [1999] study the difference in expressiveness it makes to be able to use a continuation twice. The `call/cc*` implementation of `call/cc` in terms of `call/cc-one-shot` has been proved correct several times; see §7.3 for more information.

Bruggeman et al. [1996] introduce the `call/1cc` operator used in this paper. It takes a stricter view of "one-shotness" than Friedman and Haynes [1985]: returning from `call/1cc` counts as one use of the continuation; hence, either `call/1cc` returns normally, or the captured continuation is invoked

once, but not both. Bruggeman et al. [1996] describe the implementation of `call/1cc` in terms of the segmented stack model of the Chez Scheme system, noting that no stack copying is necessary, and evaluate the performance benefits compared to regular `call/cc`. To support programs that use both `call/1cc` and `call/cc`, their implementation can promote one-shot continuations to multi-shot continuations when they are included in a multi-shot continuation captured with `call/cc`. The Iris logic of §6 also supports some combinations of `call/1cc` and `call/cc`, but it prevents multi-shot continuations from being part of one-shot continuations, since a persistent assertion such as *isCont* cannot depend on non-persistent assertions such as $isCont_1$.

Sivaramakrishnan et al. [2021] describe the design and implementation of effect handlers in OCaml 5. The continuations captured by handlers must be used *linearly*, which goes beyond the one-shot restriction: a continuation must be used exactly once, either by being invoked (`continue`) or by being explicitly discarded (`discontinue`). The authors cite three reasons for this linearity restriction: an efficient, non-copying implementation using *fibers* (multiple mutable stacks); compatibility with the explicit management of some resources (*e.g.* file descriptors); and compatibility with the optimizations performed by the OCaml compiler. In particular, OCaml can lift heap-allocated references to stack-allocated mutable variables, which would be incorrect in the presence of multi-shot continuations implemented by stack copying.

WasmFX [Phipps-Costin et al. 2023] extends WebAssembly with control operators inspired by effect handlers. The delimited continuations captured by these operators are one-shot, so that they can easily be implemented by mutable stacks. The paper mentions a possible extension to multi-shot continuations by way of a continuation clone operation based on stack copying.

When discussing control operators, we often refer to implementation considerations. Clinger et al. [1999] provide a comprehensive overview of implementation strategies for first-class undelimited continuations. Kiselyov [2012b] describes an implementation of multi-prompt delimited continuations. Xie and Leijen [2021] describes the Koka implementation of effect handlers with multi-shot continuations.

9.4 Linear continuations

In a CPS transformation or a continuation-based denotational semantics for a pure functional language, continuations are used *linearly*: every function representing a continuation has exactly one call site. Using a substructural type system, terms in CPS form can be typed while assigning linear function types to continuations. For instance, Zdancewic and Myers [2002] use linearly-typed continuations to prove that a CPS transformation preserves types for secure information flow. The result cannot be proved if continuations can be invoked multiple times.

Berdine et al. [2002] use linear types to show that continuations are used linearly in many extensions of functional languages, including mutable store, exceptions, coroutines, and some functional presentations of `goto`. The one exception is `call/cc`, which famously duplicates its continuation: $[\![\texttt{call/cc}\ e]\!] = \lambda k.\, [\![e]\!]\ k\ k$. The authors do not discuss `call/1cc`, but it would be interesting to see if their linear type system can reflect the one-shot restriction on captured continuations.

Tang et al. [2024] address the problem of combining linearly-typed resources with unrestricted effect handlers (multi-shot continuations) in the Links language. They use a linear type system to capture both value linearity and control-flow linearity. They identify interesting differences between deep and shallow effect handling from the standpoint of control-flow linearity.

Linear continuations also arise naturally in semantic accounts of classical linear logic, such as those of Filinski [1992] and Bierman [1999].

10 Conclusions and perspectives

In this paper, we constructed a separation logic for the `call/1cc` control operator — the "one-shot" restriction of the popular `call/cc` operator — and demonstrated its ability to specify and verify classic and less common uses of `call/cc` for implementing higher-level control structures and control patterns. Compared with previously proposed logics for control operators, this logic is refreshingly simple and easy to use.

This calls for a change in slogan: the `goto` of functional languages is `call/1cc`, not `call/cc`. Both `goto` and `call/1cc` are low-level, unstructured constructs that can perform arbitrary jumps in the control flow; they can express a number of more structured language constructs; they can be implemented efficiently at low implementation cost; and, as we proved in this paper, they both admit simple separation logic rules that support modular, local reasoning on their uses. In contrast, the full `call/cc` operator, with multi-shot continuations, remains in a different class than `goto` in terms of expressiveness, implementation difficulty, and the lack of local reasoning principles.

The work described in this paper can be extended in several directions. The approaches used in this paper could perhaps be extended to control operators for delimited continuations such as `shift`/`reset` and `control`/`prompt`.

The interaction between undelimited or delimited continuations, on the one hand, and exception handling and Scheme's `dynamic-wind` mechanism, on the other, has always been problematic. Developing a logic that handles both exceptions and `call/cc` could shed some light on this issue.

We might learn more by encoding effect handlers with unstructured control operators and reasoning on the encoding with an appropriate program logic, as we began to do in §8. We would like to better understand the connections with the Hazel logic of de Vilhena and Pottier [2021]. It would also be instructive to study the lexically scoped handlers of Effekt [Brachthäuser et al. 2020] and Lexa [Ma et al. 2024] from this logical angle.

Finally, our Iris formalization of `call/cc` and `call/1cc` could be reused to study existing and future type systems for these operators using the *semantic-typing* approach [Timany et al. 2024].

References

Anonymous authors. 2026. Rocq formalization. Submitted as *Supplementary Material*.

Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. 2002. Linear Continuation-Passing. *High. Order Symb. Comput.* 15, 2-3 (2002), 181–208. doi:10.1023/A:1020891112409

Martin Berger. 2009. Program Logics for Sequential Higher-Order Control. In *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5961)*, Farhad Arbab and Marjan Sirjani (Eds.). Springer, 194–211. doi:10.1007/978-3-642-11623-0_11

Gavin M. Bierman. 1999. A Classical Linear lambda-Calculus. *Theor. Comput. Sci.* 227, 1-2 (1999), 43–78. doi:10.1016/S0304-3975(99)00048-1

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. doi:10.1145/3428194

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 99–107. doi:10.1145/231379.231395

William D. Clinger, Anne Hartheimer, and Eric Ost. 1999. Implementation Strategies for First-Class Continuations. *High. Order Symb. Comput.* 12, 1 (1999), 7–45. doi:10.1023/A:1010016816429

Maurice Clint and C. A. R. Hoare. 1972. Program Proving: Jumps and Functions. *Acta Informatica* 1 (1972), 214–224. doi:10.1007/BF00288686

Tristan Crolard and Emmanuel Polonowski. 2012. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *J. Log. Algebraic Methods Program.* 81, 3 (2012), 181–208. doi:10.1016/J.JLAP.2012.01.004

Paulo Emílio de Vilhena. 2022. *Proof of Programs with Effect Handlers.* Ph. D. Dissertation. Université Paris Cité. https://theses.fr/api/v1/document/2022UNIP7133

Paulo Emílio de Vilhena. 2026. Principles of Callcc. (June 2026). Accepted for publication. https://devilhena-paulo.github.io/files/callcc.pdf

Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434314

Germán Andrés Delbianco and Aleksandar Nanevski. 2013. Hoare-style reasoning with (algebraic) continuations. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25–27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 363–376. doi:10.1145/2500365.2500593

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. doi:10.1145/1863543.1863566

Andrzej Filinski. 1992. Linear Continuations. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 27–38. doi:10.1145/143165.143174

Andrzej Filinski. 1994. Representing Monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 446–457. doi:10.1145/174675.178047

Andrzej Filinski. 1996. *Controlling Effects.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University. http://hjemmesider.diku.dk/~andrzej/papers/CE-abstract.html

Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposium on Applied Mathematics*, J. T. Schwartz (Ed.), Vol. 19. American Mathematical Society, 19–32.

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, Mary S. Van Deusen, Zvi Galil, and Brian K. Reid (Eds.). ACM Press, 245–254. doi:10.1145/318593.318654

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

Oleg Kiselyov. 2012a. An argument against call/cc. (Dec. 2012). https://okmij.org/ftp/continuations/against-callcc.html.

Oleg Kiselyov. 2012b. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.* 435 (2012), 56–76. doi:10.1016/J.TCS.2012.02.025

P. J. Landin. 1965. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM* 8, 2 (1965), 89–101. doi:10.1145/363744.363749

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2026. The OCaml system: Documentation and user's manual. https://ocaml.org/manual/5.5/index.html

Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1670–1698. doi:10.1145/3689770

Cristina Matache and Sam Staton. 2019. A Sound and Complete Logic for Algebraic Effects. In *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11425)*, Mikolaj Bojanczyk and Alex Simpson (Eds.). Springer, 382–399. doi:10.1007/978-3-030-17127-8_22

F. Lockwood Morris and Cliff B. Jones. 1984. An Early Program Proof by Alan Turing. *IEEE Ann. Hist. Comput.* 6, 2 (1984), 139–143. doi:10.1109/MAHC.1984.10017

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. doi:10.1145/3622814

Andrew M. Pitts and Ian D. B. Stark. 1999. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, Andrew D. Gordon and Andrew M. Pitts (Eds.). Cambridge University Press, USA, 227–274.

Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 118–129. doi:10.1109/LICS.2008.45

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. doi:10.1007/978-3-642-00590-9_7

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

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI 2021: 42nd ACM SIGPLAN International Conference on Programming Language Design and*

Implementation. ACM, 206–221. doi:10.1145/3453483.3454039

Kristian Støvring and Søren B. Lassen. 2007. A complete, co-inductive syntactic theory of sequential control and state. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 161–172. doi:10.1145/1190216.1190244

Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL (2024), 1600–1628. doi:10.1145/3632896

Hayo Thielecke. 1999. Using a Continuation Twice and Its Implications for the Expressive Power of Call/CC. *High. Order Symb. Comput.* 12, 1 (1999), 47–73. doi:10.1023/A:1010068800499

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. doi:10.1145/3341709

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

Alan Turing. 1949. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. Cambridge University, 67–69. Reprinted, corrected and commented in [Morris and Jones 1984]. https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/INCO.1994.1093

Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. doi:10.1145/3473576

Steve Zdancewic and Andrew C. Myers. 2002. Secure Information Flow via Linear Continuations. *High. Order Symb. Comput.* 15, 2-3 (2002), 209–234. doi:10.1023/A:1020843229247