Multiple Resumptions and Local Mutable State, Directly

SERKAN MUHCU, Technische Universität Berlin, Germany
PHILIPP SCHUSTER, University of Tübingen, Germany
MICHEL STEUWER, Technische Universität Berlin, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

While enabling use cases such as backtracking search and probabilistic programming, multiple resumptions have the reputation of being incompatible with efficient implementation techniques, such as stack switching. This paper sets out to resolve this conflict and thus bridge the gap between expressiveness and performance. To this end, we present a compilation strategy and runtime system for lexical effect handlers with support for multiple resumptions and stack-allocated mutable state. By building on garbage-free reference counting and associating stacks with stable prompts, our approach enables constant-time continuation capture and resumption when resumed exactly once, as well as constant-time state access. Nevertheless, we also support multiple resumptions by copying stacks when necessary. We practically evaluate our approach by implementing an LLVM backend for the Effekt language. A performance comparison with state-of-the-art systems, including dynamic and lexical effect handler implementations, suggests that our approach achieves competitive performance and the increased expressiveness only comes with limited overhead.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Control structures**; Functional languages, Imperative languages; • **Theory of computation** → **Control primitives**.

Additional Key Words and Phrases: algebraic effects, effect handlers, multiple resumptions, local mutable state, stack switching, garbage-free reference counting

ACM Reference Format:
Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. 2025. Multiple Resumptions and Local Mutable State, Directly. *Proc. ACM Program. Lang.* 9, ICFP, Article 260 (August 2025), 30 pages. https://doi.org/10.1145/3747529

1 Introduction

A contemporary way to structure advanced patterns of control-flow transfers is by using effect handlers [Plotkin and Pretnar 2013]. In the last decade, effect handlers have seen a growing adoption in research languages (such as Eff [Plotkin and Pretnar 2013], Frank [Lindley et al. 2017], Koka [Leijen 2017b], Effekt [Brachthäuser et al. 2020], Helium [Biernacki et al. 2019], Lexa [Ma et al. 2024], Flix [Lutze and Madsen 2024], and more) as well as in industrial languages like OCaml [Sivaramakrishnan et al. 2021] and WebAssembly [Phipps-Costin et al. 2023]. Effect handlers decouple the use of effects in effectful programs from their concrete implementation in an effect handler. Calling an effect operation transfers control from its use site to the corresponding handler and captures the delimited continuation up to the handler. A handler can then decide to not resume the continuation (*e.g.*, for exceptions), resume it exactly once in tail position (*e.g.*, for dynamic binding), resume it exactly once but later (*e.g.*, for async/await), or resume it multiple times.

Authors' Contact Information: Serkan Muhcu, Technische Universität Berlin, Berlin, Germany, serkan.muhcu@tu-berlin.de; Philipp Schuster, University of Tübingen, Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Michel Steuwer, Technische Universität Berlin, Berlin, Germany, michel.steuwer@tu-berlin.de; Jonathan Immanuel Brachthäuser, University of Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de.

ACM 2475-1421/2025/8-ART260
https://doi.org/10.1145/3747529

Multiple Resumptions. Indeed, the ability to resume multiple times (*i.e.*, multi-shot continuations in contrast to one-shot continuations that can only be resumed at most once), is a very useful language feature that facilitates implementations of logic programming [Haynes 1987; Saleh and Schrijvers 2016], probabilistic programming [Goldstein and Kammar 2024; Kiselyov and Shan 2009; Phan et al. 2019], parsing [Leijen 2016], backtracking search [Friedman et al. 1984], and more.

However, being able to resume continuations multiple times seemingly precludes efficient implementation. In consequence, languages like OCaml choose to not support it.

> *This example does not resume a continuation more than once. This also holds true for other use cases such as generators and coroutines. Hence, our continuations are one-shot, and resuming the continuation more than once raises an* `Invalid_argument` *exception. It is well-known that one-shot continuations can be implemented efficiently.*
> — [Sivaramakrishnan et al. 2021]

Similarly, recent proposals for WebAssembly choose to not support it.

> *[…] an extension to support multi-shot continuations would be interesting, but difficult to support efficiently or robustly in existing Wasm engines.*
> — Phipps-Costin et al. [2023]

Others, like Java's implementation of lightweight fibers, officially only support one-shot continuations, but allow the programmer to manually copy the continuation, which is fragile and non-modular for nested handlers.

> *Loom currently implements what can be called "[one-shot] asymmetric delimited continuations with multiple named prompts," but with cloning, you get [multi-shot] delimited continuations.* — Pressler [2018]

Industry-grade implementations that support multiple resumptions, like GHC's [King 2022] or Scala Native's [Pham and Odersky 2024], defensively copy the continuation, incurring a cost even for one-shot usage of the continuations. To efficiently support multi-shot continuations and avoid copying, some implementations alternatively represent the runtime stack as an immutable data structure [Brachthäuser et al. 2018; Farvardin and Reppy 2020]. However, this implementation strategy seems fundamentally incompatible with local, *i.e.* stack-allocated, mutable state.

Local Mutable State. Mutable state is believed to interact badly with multiple resumptions.

> *Whether to allow continuations to be multi-shot has far-reaching consequences related to performance and program reasoning, especially in languages with mutable references. […] Certain standard program transformations (and thus compiler optimisations) are unsound in the presence of multi-shot continuations because the rules of reasoning that justify them no longer hold* — van Rooij and Krebbers [2025]

Continuations are parts of the runtime stack including stack-allocated mutable references. If resumed twice, the first resumption might update references and affect the second resumption. Implementations that support mutable references and multiple resumptions, like Koka and Effekt, create a backup of the state on capturing the continuation and restore it upon resumption, negatively impacting performance. If the continuation is used at most once, the backup is unnecessary.

Stack Switching for Lexical Effect Handlers. There are two main variations of effect handlers:

- **Dynamic Effect Handlers**. Traditionally, like exceptions, effect handlers are dynamically scoped. Calling an effect operation entails unwinding the call stack to both search for the correct handler and to capture the continuation. This form of effect handling is implemented in Eff [Plotkin and Pretnar 2013], Koka [Leijen 2017b], Frank [Lindley et al. 2017], and OCaml [Sivaramakrishnan et al. 2021].

- **Lexical Effect Handlers**. Alternatively, a lexical connection can be established between an effect handler and the use of an effect operation. This way, calling an effect operation reduces to a dynamic dispatch followed by unwinding the call stack to capture the continuation. This form of effect handling is implemented in Effekt [Brachthäuser et al. 2020], Helium [Biernacki et al. 2019], Lexa [Ma et al. 2024], and Scala [Pham and Odersky 2024].

It has been shown recently that lexical effect handlers enable a low-level implementation of constant-time continuation capture and resumption by passing a direct pointer to a stack segment, instead of linearly searching for a handler or a marker [Ma et al. 2024]. However, their approach seems incompatible with multiple resumptions.

Contributions. Our goal is an efficient implementation of lexical effect handlers supporting **multiple resumptions** and **local mutable references**, with the following properties:

- correct and predictable interaction between those two
- constant-time capture and resume for one-shot use of continuations
- constant-time read and write for local mutable references

Reconciling these conflicting goals seems difficult. How do we know which references to backup and restore, if at all? Can we know upfront how often a continuation is resumed? How can we avoid defensive copying? In this paper, we generalize the approach of Ma et al. [2024] to support multi-shot continuations, while retaining good performance characteristics. Our solution rests on the following key observations:

Stable prompts. To enable constant-time stack switching and state access, we need to be able to access stack segments directly, without traversal. This can be achieved by using pointers to stacks. However, multiple resumptions require stack copying, which invalidates any pointers to this stack. We introduce prompts with stable addresses as an indirection to stacks. This allows copying stacks, while only having to update a single pointer, and pointers to the stable prompts remain valid.

Reference counting. To efficiently support resuming exactly once, we need to avoid copying. To this end, we use garbage-free reference counting [Reinking et al. 2021], which provides us with an up-to-date reference count. This way we only copy whenever we resume a shared continuation.

While these core ideas seem appealingly simple, they require a carefully designed runtime system, which we formalize in Section 3. Furthermore, using reference counting also comes with its price: we have to keep the reference counts up-to-date. Consequently, while we support a one-shot use of continuations (*e.g.* async / await) in constant time, discarding the continuation (*e.g.* exceptions) requires us to unwind the continuation in linear time. Moreover, the additional indirection over the prompt incurs an additional minimal overhead per stack switching operation.

1.1 Contributions

We present the design and implementation of a runtime system for multi-shot delimited control and local mutable references by making the following contributions:

- We introduce key ideas based on an example (Section 2). Our design and implementation additionally supports first-class functions, proper tail calls, effect-polymorphic recursion, growable stacks, region-based memory, and bidirectional handlers.
- We formalize the compilation as well as the runtime system as a translation from a low-level intermediate language MCode with support for multiprompt delimited control and local mutable state to an instruction set ASM (Section 3).
- We assess the practical consequences of our approach with an implementation of a novel backend for the Effekt language, targeting LLVM (Section 4). Our new backend is feature

complete and successfully compiles the standard library spanning around 10k lines of code as well as another 10k lines of code of tests.
- We evaluate the performance of our implementation by comparing to other state-of-the-art implementations on community benchmarks (Section 5). Results suggest that providing the additional expressivity of multiple resumptions only comes with limited cost compared to the implementation by Ma et al. [2024], while it outperforms other implementations on average.

1.2 Limitations

Our approach has some limitations:

- There is an additional indirection when accessing local mutable references, which leads to a constant-time overhead. We assess this overhead in Section 5. Our measurements suggest that it is insignificant in benchmarks for effect handlers, but significant in loops that heavily use local mutable references.
- Resumptions are not *reentrant*, meaning that a continuation cannot be resumed, while it is already running. Our implementation leads to a runtime error in this case. We discuss this limitation in Section 6.

We begin by introducing our approach by example (Section 2), before we formalize (Section 3) and evaluate it (Sections 4 and 5).

2 Multi-shot Stack Switching by Example

In this section, we demonstrate multiple resumptions, local mutable state, and their non-trivial interaction. We present examples in Effekt [Brachthäuser et al. 2020], a general-purpose programming language with lexical effect handlers. We then illustrate the key insights behind our runtime system with efficient support for both features. Later, in Section 3, we will revisit the examples more formally in a low-level intermediate representation.

Simple effects and mutable state. Effectful programs are those that interact with their context non-trivially. They stand in contrast to pure programs, whose only interaction with their context is by returning a result when they are done. *Effect signatures* specify the shape of these interactions between programs and their contexts. This is in analogy with how types specify the shape of data. For example, consider the following effect signature for generators of integer numbers.

```
effect emit(v: Int): Unit
```

It specifies that the program emits values of type `Int` to its context and receives results of type `Unit` in return.

Effectful functions use effects according to their signatures. In addition to their parameter and return types, we track their effects. For example, we can generate a stream of integer numbers.

```
def range(l: Int, h: Int): Unit / emit = if (l < h) { do emit(l); range(l + 1, h) }
```

The effectful function `range` receives a lower and an upper bound of type `Int`, returns a value of type `Unit`, and uses an effect `emit`. As long as `l < h`, we `do emit(l)` and recurse.

The meaning of this function depends on the context it is called in. Indeed, it must emit the integer numbers somewhere. *Effect handlers* provide this meaning when they are part of the program's context. For example, we can compute the sum of all values emitted by a given stream.

```
def sum { stream: () ⇒ Unit / emit }: Int = {
  var s = 0;
  try stream() with emit { v ⇒ s = s + v; resume(()) };
  return s }
```

To do so, we allocate a *local mutable reference* `s`, initialized to `0`. We then run the given program `stream` under a handler for the `emit` effect. For each emitted value `v`, we increase the state of `s` and resume the program with the unit value `()`. Finally, we return the state of `s`.

The local mutable reference `s` shall not be used outside the body of `sum`. This way, the mutable state is encapsulated, and purity is preserved. We can allocate the value of `s` on the runtime stack, avoiding overhead. Functions and handlers can close over local mutable references like `s`. The escaping of local mutable references can be prevented in the same way general effect safety is achieved. In fact, `var s = 0` can be thought of as a small handler for mutable state.

Finally, we compute the sum of a stream of numbers by composing the effectful program with the handler function: `sum {range(1,5)}`. The effectful program `range(1,5)` repeatedly emits a value to the effect handler installed in `sum`, which updates the local mutable reference `s` and resumes. This interaction is governed by the effect signature `emit`. While the handler function `sum` uses a mutable reference locally, the overall program is pure from the outside. This is a common pattern. Handler functions accumulate information and keep state using local mutable references.

Resuming multiple times. In the previous example, the handler for `emit` resumed the continuation exactly once after increasing `s` with the received value. In general, handlers can choose to never resume the continuation, resume it later, or resume it multiple times. The latter results in an interesting interaction with local mutable references. To illustrate, let us now consider a more advanced use of effect handlers: checkpointing. Here, we use two effect operations, `save` and `retry`, for saving the current program state and rolling it back.

```
effect save(): Unit          effect retry(): Nothing
```

The following program uses these effects. It asks the user to enter a stream of numbers and provides the ability to commit the data entered so far and to undo everything up until the last commit.

```
def user(): Unit / {emit, save, retry} = input() match {
  case Enter(n) ⇒ do emit(n); user()
  case Commit() ⇒ do save(); user()
  case Undo() ⇒ do retry()
  case Done() ⇒ () }
```

When the user enters a number `n`, we emit it. When comitting the current state, we use `save`, and when the user wants to undo changes, we use `retry`. The following handler now handles the two operations, `save` and `retry`, by installing a checkpoint and retrying from the last one, respectively.

```
def checkpointing[R]{ program: () ⇒ R / {save, retry} }: R = {
  var c = None();
  try program()
  with save { c = Some(resume); resume(()) }
  with retry {
    if (c is Some(resumption)) { resumption(()) }
    else { checkpointing{program} } } }
```

We first allocate a local mutable reference `c` for holding the last saved resumption, initially empty. When the program uses `save`, we overwrite the saved resumption with the current one, `Some(resume)`, and then resume the program. When the program uses `retry`, we check if there is a saved resumption, and if there is, we resume from that point. If there is not, we restart the program from the beginning. In neither case do we use `resume(())`, since we want to abort the current computation and resume from an earlier point. This handler, depending on the handled program, will thus invoke some resumptions once, zero times, or multiple times.

Now, we can compute the sum of user inputs with checkpointing and rollback by executing the program `user()` under the two handler functions, `sum` and `checkpointing`.

```
checkpointing { sum { user() } }
```

For example, inputs `Enter(1), Commit(), Enter(2), Undo(), Enter(3), Done()` result in the total sum `1 + 3`, because the entry of the number `2` is rolled back. For this to work, we have to back up the current state of the local mutable reference `s`, which is hidden in `sum`. More generally, checkpoints and continuations, where it can only be known at runtime how often resumption happens, are a compelling use case for multi-shot continuations [Koskinen and Herlihy 2008].

To backtrack or not to backtrack. Not every mutable reference that exists in the program state should be rolled back when a continuation is resumed for the second time. Consider the following program, where we create an outer mutable reference `i`, install a checkpointing handler, and create an inner mutable reference `s`. We repeatedly retry from the save point, incrementing both references.

```
var i = 0;
checkpointing {
  var s = 0;
  do save();
  println(s);
  i = i + 1; s = s + 1;
  if (i < 4) { do retry() } }
```

Possible outcomes.

(1) The program prints `0`, `0`, `0`, `0` and halts;
(2) The program prints `0`, `1`, `2`, `3` and halts;
(3) The program prints `0`, `0`, `0`, `0` … forever; or
(4) The program prints `0`, `1`, `2`, `3` … forever.

When executing this program, depending on the backtracking behavior of both mutable references, one of four things (on the right) can conceivably happen. We follow Kiselyov et al. [2006] and argue that behavior (1), where `s` is backed up and restored but `i` is not, is the desired one. From the point of view of `checkpointing`, `i` is a global mutable reference, while `s` is a local mutable reference. Operationally, upon `save`, the captured continuation is delimited by the handler `checkpointing` and thus contains a copy of the state of `s` but not a copy of the state of `i`.

This backtracking behaviour enables local reasoning about the state of local mutable references, which is desirable in many cases. Take the following example, where we write to a local mutable reference `x` between two function calls.

```
var x = 0; f(x); x = x + 1; f(x)          f(0); f(1)
```

With multiple resumptions, the first call to `f` might return multiple times, each time incrementing `x`. But since the state of `x` is backtracked, the program is still equivalent to the one on the right.

This is in contrast to global mutable references, whose state is not preserved across resumptions. Effekt offers those in its standard library. Here is the same example using a global mutable reference:

```
val x = ref(0); f(x.get()); x.set(x.get() + 1); f(x.get())
```

Using the library offers a clear syntactic distinction between local and global mutable references.

Existing implementations. Multiple resumptions and backtrackable mutable state seem to be at odds with efficient implementation strategies like stack switching [Ma et al. 2024; Phipps-Costin et al. 2023; Sivaramakrishnan et al. 2021], where capturing and resuming a continuation merely amounts to changing a few pointers. In cases where a continuation is resumed at most once, it seems fine to destructively update the state allocated on the stack. However, if we resume multiple times, this would not align with the desired backtracking semantics described above.

As a consequence, implementations of effect handlers and mutable state either do not support multiple resumptions [Ma et al. 2024; Phipps-Costin et al. 2023; Sivaramakrishnan et al. 2021] or

perform defensive copying (like Effekt and Koka). In this paper, we show that it is indeed possible to reconcile multiple resumptions and backtrackable mutable state with stack switching.

2.1 Simple Stack Switching by Example

To illustrate our approach, let us start by inspecting how stack switching works in the simpler example program `sum { range(1, 5) }`. As a first step, the Effekt compiler translates the program into an intermediate representation based on System C, in capability-passing style and using multi-prompt delimited control [Brachthäuser et al. 2022; Brachthäuser et al. 2020].

```
sum { {e} ⇒ range(1, 5){e} }

def range(l: Int, h: Int){e: emit} =
  if (l < h) {
    e.emit(l);
    range(l + 1, h)
  }
```

```
def sum { stream: {emit} ⇒ Unit }: Int = {
  var s = 0;
  reset { p ⇒
    def e = new emit { v ⇒
      shift(p) { k ⇒ s = s + v; k(()) } }
    stream{e} }
  return s }
```

Importantly, the two aspects of handling an effect are translated into two different concepts. Firstly, a *capability* `e` is created as an instance of `emit`, containing the handler implementation. This capability is then explicitly passed to functions that previously used the `emit` effect, such as `range`. Secondly, the capability can capture the continuation by means of `shift`. The captured continuation is delimited by the corresponding `reset`. The connection between `reset` and `shift` is established by a freshly introduced *prompt* `p`.

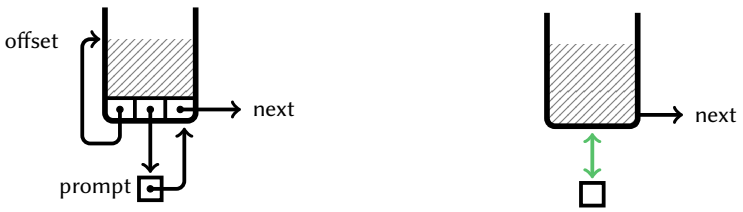


Fig. 1. Graphical representation of an individual stack segment (left) and its short-hand notation (right). The stack is *valid*, that is, the prompt points back to the stack itself.

In our stack-switching implementation, programs run with a *meta stack*, a linked list of individual call stacks. Operationally, `reset` introduces a fresh call stack and links it to the current meta stack. In addition to the new stack segment, `reset` also introduces a fresh *prompt*, which is a heap-allocated mutable cell that points back to the stack. In the following, we use the graphical notation from Figure 1. We refer to stacks where the prompt points back to the stack itself as *valid* and depict them using the shorthand notation on the right, with a green double-headed arrow.

Constant-time continuation capture. Figure 2 (left) shows our meta stack when calling `e.emit(1)`, that is, after evaluating the `reset` and entering the function `range`, just before the call to `shift(p)`. Programs run with a meta stack pointer `msp`, which points to the first stack. In this example, it contains the frame `range(2, 5)`, which is the delimited continuation in `range` after the call to `emit`. The next stack in the list contains a frame returning the value of `s`, and then the a frame that contains the current state of `s`, which is `0`. Capturing the continuation by switching stacks now amounts to the following steps:

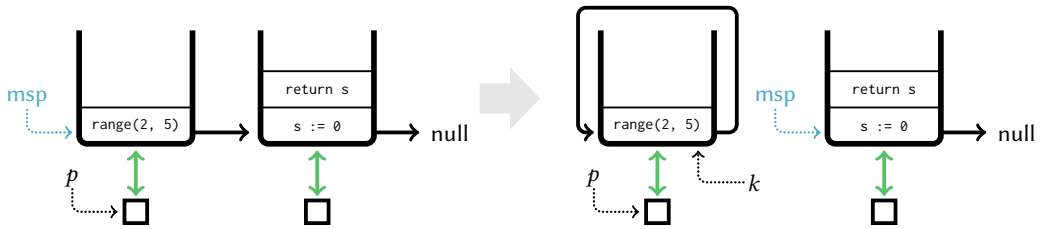


Fig. 2. Capturing the continuation k up to (and including) prompt p.

(1) We dereference prompt p, finding that (in this particular case) it points to the first stack segment. In general, we can capture arbitrarily many stack segments.
(2) We swap `msp` and `next` so that the meta stack pointer points to the next stack segment and `next` creates a cycle. In our approach, continuations always have this cyclic form.

We can notice that the continuation is valid, as for all contained stack segments (in this case only one), the prompts point back to the segment itself. As we will see, in our approach, a continuation can be in one of two states: either it is valid, as described above, or it is *invalid*, which means that for *all* stack segments, the prompt cell points somewhere other than the stack segment itself.

Constant-time continuation resumption. The following picture in Figure 3 (left) illustrates the state of program execution after increasing `s` by the emitted value 1 and right before resuming.

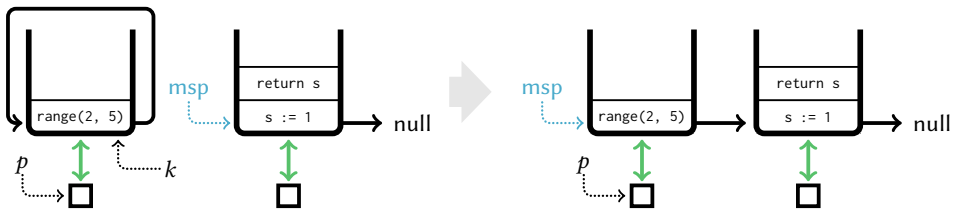


Fig. 3. Resuming continuation k.

Before we resume a continuation, we inspect its reference count to determine if we need to copy it or not. Since we rely on up-to-date reference counts, garbage-free reference counting [Reinking et al. 2021] is crucial to our approach. In this particular case, we have a unique reference to k. We then check whether the continuation is valid. Since continuations are either valid or invalid, it suffices to check one stack for validity. In this particular case, the continuation is valid. We resume by swapping the two pointers `next` and `msp`, as illustrated. For this one-shot use of the continuation, resuming thus only requires 3 loads, 1 store, and 1 pointer comparison. Importantly, this is independent of the size of the continuation: capturing and resuming can be performed in constant time, independent of the number of handlers in between.

Constant-time backtrackable mutable state. In the example above, we destructively updated the state of the stack-allocated mutable reference. In our implementation, local mutable references are represented as pairs of a prompt and an offset on the stack. Reading a mutable reference amounts to first loading the base address of the corresponding stack segment from the prompt, followed by a second load relative to the base address. Reading and writing mutable references thus also occur in constant time, independent of the number of handlers in between.

2.2 Multiple Resumptions

So far, our approach does not seem very different from existing stack-switching implementations, which only work for one-shot continuations. To generalize to multi-shot continuations, we add three key ingredients: (1) checking the reference count on the continuation to determine whether the reference is unique, (2) using prompts as separate mutable references that point to stacks, and consequently, (3) checking whether the continuation is valid before resuming. While unnecessary for one-shot continuations, we will see why these are the key enablers for multi-shot continuations.

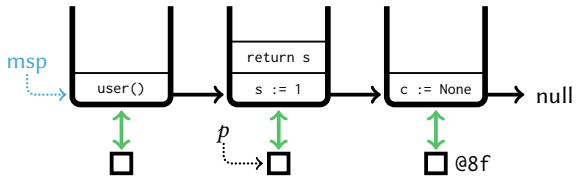


Fig. 4. Before capturing the continuation.

To this end, let us consider our example `checkpointing { sum { user() } }` after processing `Enter(1)` in `user()`, in the case of `Commit`, when performing the call to `do save()`. In this example, shown in Figure 4, we have three stack segments, which correspond (from right to left) to the top-level program, the delimiter installed by `checkpointing`, and the delimiter installed by `sum`. Since we have not stored any continuation, `c` is initialized with `None`, and since we already processed `Enter(1)`, the running total `s` is 1. To handle the call to `save`, we invoke the body `shift(p) { k ⇒ c = Some(k); k(()) }` of the corresponding capability. Capturing the continuation now proceeds as described above.

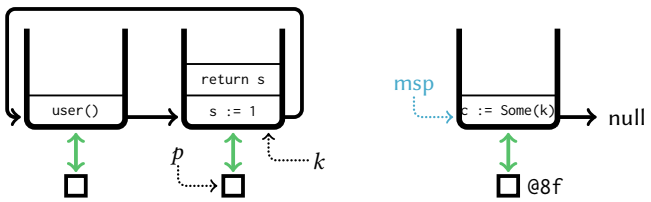


Fig. 5. After capturing continuation k and storing it in `c`.

We successfully captured the continuation, stored it in `c`, and increased its reference count, as shown in Figure 5. Next, we now want to resume the continuation `k(())`. Checking the reference count, we notice that we are not the unique owner. Proceeding as previously and directly switching back to the continuation would lead to destructive updates of the involved stack segments. Since we are not the sole owner of the continuation, we create a copy, as shown on the left in Figure 6.

Noticeably, the copy k' has the same structure, and the stack segments point to the same prompts as the original k. However, those prompts do not point back to the copy, and hence the copy is not valid, depicted with a red single-headed arrow. In order to resume the copy, we need to *revalidate* it, the result of which can be seen on the right-hand side. Revalidating the copy k' invalidates the original k. Since the copy k' is now both unique and valid, we can resume it exactly as in the previous section. The next entered value, 2, will lead to a destructive update of state s in this copy, while the original continuation k stays unchanged. Consequently, the handler implementation of `retry`, upon user input `Undo()`, will restore another fresh copy of k, which again starts off with `s` having value 1, implementing the correct backtracking behavior.

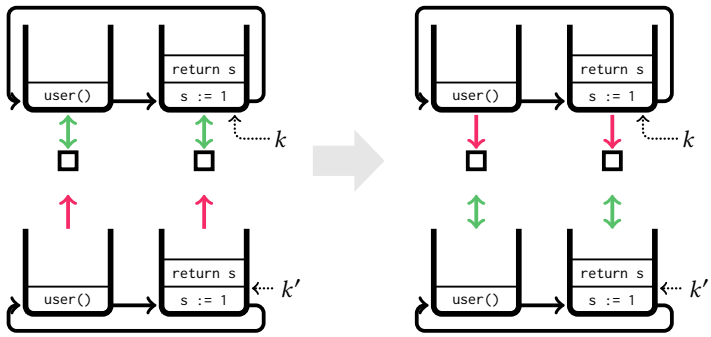


Fig. 6. Resuming a shared continuation requires copying it (result shown left). Resuming k' requires revalidating (result shown right).

2.3 Section Conclusion

We have seen how multiple resumptions and mutable state interact and how useful behavior emerges from this interaction. To reconcile efficient stack switching with multiple resumptions requiring copying, our implementation technique rests upon three key aspects. Firstly, we introduce separate prompt cells with stable addresses. Individual stack segments do not need a stable address and can be copied. Secondly, we use the reference count of a continuation to determine whether we are resuming it for the last time (or possibly the only time). If not, we fall back to copying the continuation. Thirdly, we require prompts to point to the active stack segments. To maintain this invariant, each stack segment keeps a pointer to the corresponding prompt. This way we support constant-time continuation capture, state access and modification, and continuation resumption in the one-shot case while being prepared to resume multiple times when necessary.

3 Formalization

In this section, we formalize our approach to compilation of and runtime support for delimited control operators and local mutable references with efficient stack switching. To focus on the essentials, we start our formal treatment with a stripped-down language MCode that includes delimited control operators and local mutable references but not much more. To illustrate the operational behavior on contemporary hardware, we then define the compilation from MCode to an idealized machine instruction set ASM. The compiled code uses runtime system features for delimited control operators and local mutable references, which we also define. We then discuss the time complexity of these features.

3.1 Source Language MCode

Figure 7 lists the syntax of our source language MCode. Since our goal is the formalization and explanation of operational behavior, it is untyped, and we neither guarantee type nor effect safety. Moreover, it is not intended for direct use by programmers but rather as an intermediate representation. It does not feature nested expressions, and free variables in closures and pushed frames must be explicit. We use the meta variable f to denote known top-level function definitions, in contrast to the meta variable x, which stands for values created at runtime. Moreover, by convention, instead of x, we use c for closures, p for prompts, k for continuations, and r for references.

A program is a list of potentially mutually recursive top-level definitions with name f, parameters $\overline{x}$, and body s. We perform a direct jump with arguments to f using **jump** $f(\overline{x})$. These jumps are always tail calls that do not touch the stack. When we require a different return context, we must

Programs:

P	::=	$\overline{\textbf{def}\ f(\overline{x})\ =\ s}$	Function definitions

Statements:

s	::=	$\textbf{jump}\ f(\overline{x})$	Direct jump
	|	$\textbf{push}(\overline{x})\{\ x \Rightarrow s\ \};\ s$	Push stack frame
	|	$\textbf{return}\ x$	Pop and run topmost stack frame
	|	$c\ =\ \textbf{new}(\overline{x})\{\ (\overline{x}) \Rightarrow s\ \};\ s$	Closure allocation
	|	$\textbf{call}\ c(\overline{x})$	Indirect jump
	|	$p\ =\ \textbf{reset};\ s$	Delimit computation
	|	$k\ =\ \textbf{shift}\ p;\ s$	Capture delimited continuation
	|	$\textbf{resume}\ k;\ s$	Resume continuation
	|	$r\ =\ \textbf{ref}\ x;\ s$	Allocate mutable reference
	|	$x\ =\ \textbf{get}\ r;\ s$	Load value from reference
	|	$\textbf{set}\ r\ x;\ s$	Store value into reference

Variables:

f	Functions
x	Values
c	Closures
p	Prompts
k	Continuations
r	References

Fig. 7. Syntax of programs and statements in MCode. Free variables in frames and closures are explicit.

explicitly push a frame with $\textbf{push}(\overline{y})\{\ x \Rightarrow s\ \}$. Only the frame environment $\overline{y}$ and the returned value x are available in the body s. We return to the most recently pushed frame with **return** x. Similarly, when we create a closure with $c\ =\ \textbf{new}(\overline{y})\{\ (\overline{x}) \Rightarrow s\ \}$, only the closure environment $\overline{y}$ and the parameters $\overline{x}$ are available in the body s. Calls to closures $\textbf{call}\ c(\overline{x})$ are indirect tail calls; for non-tail positions, we must push a frame first.

The language constructs discussed so far are fairly standard. Let us now turn to delimited control operators and local mutable references. We install a delimiter with $p\ =\ \textbf{reset};\ s$. In return, we receive a prompt p, which marks this position in the context and is available in the delimited statement s. We shift to a marked position with $k\ =\ \textbf{shift}\ p;\ s$. This gives us a continuation k in statement s. We can then resume this continuation with $\textbf{resume}\ k;\ s$, where s runs in the context of the reinstalled continuation k. Prompts are first-class but always created freshly and associated with one specific delimiter (similar to *spawn/controller* by Hieb and Dybvig [1990]). We allocate a local mutable reference r with initial value x that is available in statement s with $r\ =\ \textbf{ref}\ x;\ s$. Like prompts, references are first-class but only safe to use in the dynamic extent of the **ref** operator.

Example. In the following, we illustrate how we translate the motivating example from Section 2, which already is in capability-passing style, to MCode, making explicit frames and closures.

```
def range(l, h, emit) =
  if (l < h) {
    push(l, h, emit) { _ ⇒
      jump range(l + 1, h, emit)
    };
    call emit(l)
  } else {
    return ()
  }
```

```
def sum(stream) =
  s = ref 0;
  push(s) { _ ⇒
    x = get s; return x };
  p = reset;
  emit = new(p, s) { (v) ⇒
    k = shift p;
    x = get s; set s (x + v);
    resume k; return () };
  call stream(emit)
```

```
stream = new() { (emit) ⇒
  jump range(1, 5, emit)
};
jump sum(stream)
```

The effectful function range emits a stream of integer numbers. In explicit capability-passing style, it not only receives l and h, but also a capability emit for doing so. The capability is an ordinary function. Before calling it, we push a frame that remembers to generate the rest of the values. The handler function sum provides the capability emit to the given program stream. This capability closes over the freshly installed prompt p and the freshly allocated reference s. It shifts to the prompt, updates the state, and resumes. After returning through the delimiter, but before deallocating the reference, we get the final value x, which we return as the overall result. To compose the handler function with the effectful program, we create a closure stream that receives the emit capability and jumps to range. We then jump to sum with this closure argument. Notably, these examples are written in explicit capability-passing style using delimited control operators and local mutable references. We distinguish between direct jumps and indirect calls. All jumps and calls are tail calls, and frames have to be pushed explicitly.

Tail-resumption optimization. One trivial but common usage pattern of effect handlers is to resume exactly once in tail position. Such a *tail resumptive* handler effectively expresses dynamic binding [Brachthäuser and Leijen 2019]. To improve performance, languages like Koka [Leijen 2017a; Xie and Leijen 2021] or Effekt [Brachthäuser 2024] optimize these handlers to avoid capturing the continuation altogether. The optimization is typically syntactic and does not cover other one-shot usages of the continuation. We observe that in this example, in the body of emit, we shift and then resume exactly once in tail position, and could optimize emit to not shift and resume at all.

```
emit = new(s) { (v) ⇒ x = get s; set s (x + v); return () };
```

This optimization is local and does not affect the rest of the program. It is valid since, thanks to the lexicality of prompts and references, we know that s still refers to the same reference. It is local because we perform capability passing. Capabilities are closures that are always called in the same way, whether they use effects or not.

3.1.1 Semantics. Figure 8 defines the operational semantics of MCode as an abstract machine. Machine configurations consist of the statement s under execution, environment E, stack K, and program P. Environments map variables to values v, and stacks are lists of frames F. Values and frames contain markers m, which are freshly generated at runtime. Values are closures with environment E, parameters $\overline{x}$, and body s, continuations with delimiter m and stack K, or markers m. As we will see, we use marker values for both prompts and references. Frames are calling contexts $\{\ E,\ x \Rightarrow s\ \}$, delimiters **reset** m, or references **ref** $m\ v$ with current state v. The first set of reduction rules is completely unsurprising. We omit program P from all rules except **jump**.

When we execute p = **reset**, we generate a fresh marker m, which we push onto the stack and bind to p. This installs a delimiter with marker m. When we execute k = **shift** p, we look up the marker m that p stands for. We split the stack at this marker, bind the prefix as continuation k, and keep the suffix as the stack. When we then execute **resume** k, we simply restore the delimiter and stack. Finally, when we return through a delimiter, we discard it and return to the next frame.

When we execute r = **ref** y, we generate a fresh marker m and allocate a reference with m and store the current value of y on the stack. We bind the reference r to this marker. When we execute x = **get** r, we find the state v corresponding to the marker m of reference r on the stack. When we execute **set** r y, we replace the state at marker m of reference r with the value of y. Finally, when we return through a reference on the stack, we discard it and return to the next frame.

When a marker for a delimiter or a reference is not found on the stack, we leave the behavior undefined in this semantics. It is the responsibility of a high-level language, possibly with a type-and-effect or region system, to rule out these cases [Brachthäuser et al. 2020; Schuster et al. 2022].

Configurations:

$M \quad ::= \quad \langle\, s \mid E \mid K \mid P \,\rangle$

Environments:

$E \quad ::= \quad \overline{x \mapsto v}$

Stacks:

$K \quad ::= \quad F :: K \mid \bullet$

Markers:

$m \quad ::= \quad$ `@a5f` | `@4b2` | ...

Values:

$v \quad ::= \quad \{\, E,\ (\overline{x}) \Rightarrow s \,\} \mid \textbf{cont}\ K\ m \mid m$

Frames:

$F \quad ::= \quad \{\, E,\ x \Rightarrow s \,\} \mid \textbf{ref}\ m\ v \mid \textbf{reset}\ m$

Machine Steps:

$\langle\, \textbf{jump}\ f(\overline{y}) \mid E \mid K \mid P \,\rangle \rightarrow \langle\, s \mid \overline{x \mapsto E(y)} \mid K \mid P \,\rangle$ where $\textbf{def}\ f(\overline{x}) = s \in P$

$\langle\, \textbf{push}\ (\overline{y})\ \{\, x \Rightarrow s_0 \,\};\ s \mid E \mid K \,\rangle \rightarrow \langle\, s \mid E \mid \{E_0,\ x \Rightarrow s\} :: K \,\rangle$ where $E_0 = \overline{y \mapsto E(y)}$

$\langle\, \textbf{return}\ y \mid E \mid \{\, E_0,\ x \Rightarrow s \,\} :: K \,\rangle \rightarrow \langle\, s \mid E_0,\ x \mapsto v \mid K \,\rangle$ where $v = E(y)$

$\langle\, c = \textbf{new}\ (\overline{y})\ \{\, (\overline{x}) \Rightarrow s_0 \,\};\ s \mid E \mid K \,\rangle \rightarrow \langle\, s \mid E,\ c \mapsto \{E_0,\ (\overline{x}) \Rightarrow s_0\} \mid K \,\rangle$ where $E_0 = \overline{y \mapsto E(y)}$

$\langle\, \textbf{call}\ c(\overline{y}) \mid E \mid K \,\rangle \rightarrow \langle\, s \mid E_0,\ \overline{x \mapsto E(y)} \mid K \,\rangle$ where $E_0,\ (\overline{x}) \Rightarrow s = E(c)$

Delimited Control

$\langle\, p = \textbf{reset};\ s \mid E \mid K \,\rangle \rightarrow \langle\, s \mid E,\ p \mapsto m \mid \textbf{reset}\ m :: K \,\rangle$ where m fresh

$\langle\, k = \textbf{shift}\ p;\ s \mid E \mid K_1 :: \textbf{reset}\ m :: K_2 \,\rangle \rightarrow \langle\, s \mid E,\ k \mapsto \textbf{cont}\ K_1\ m \mid K_2 \,\rangle$ where $m = E(p)$

$\langle\, \textbf{resume}\ k;\ s \mid E \mid K_2 \,\rangle \rightarrow \langle\, s \mid E \mid K_1 :: \textbf{reset}\ m :: K_2 \,\rangle$ where $\textbf{cont}\ K_1\ m = E(k)$

$\langle\, \textbf{return}\ y \mid E \mid \textbf{reset}\ m :: K \,\rangle \rightarrow \langle\, \textbf{return}\ y \mid E \mid K \,\rangle$

Mutable State

$\langle\, r = \textbf{ref}\ y;\ s \mid E \mid K \,\rangle \rightarrow \langle\, s \mid E,\ r \mapsto m \mid \textbf{ref}\ m\ E(y) :: K \,\rangle$ where m fresh

$\langle\, x = \textbf{get}\ r;\ s \mid E \mid K_1 :: \textbf{ref}\ m\ v :: K_2 \,\rangle \rightarrow \langle\, s \mid E,\ x \mapsto v \mid K_1 :: \textbf{ref}\ m\ v :: K_2 \,\rangle$ where $m = E(r)$

$\langle\, \textbf{set}\ r\ y;\ s \mid E \mid K_1 :: \textbf{ref}\ m\ v :: K_2 \,\rangle \rightarrow \langle\, s \mid E \mid K_1 :: \textbf{ref}\ m\ E(y) :: K_2 \,\rangle$ where $m = E(r)$

$\langle\, \textbf{return}\ y \mid E \mid \textbf{ref}\ m\ v :: K \,\rangle \rightarrow \langle\, \textbf{return}\ y \mid E \mid K \,\rangle$

Fig. 8. Syntax and stepping relation of the MCode abstract machine.

Similarly, when a marker is on the stack twice, we leave the behavior undefined. It is again the responsibility of the high-level language to rule out these cases.

3.2 Target Language ASM

Figure 9 defines the syntax of our target language ASM. It abstracts over some details that would be expected in a low-level representation to focus on describing how we compile continuation capture, resumption, and mutable state. Words w can be integers, pointers, or null. Registers r map to words, and we assume an infinite number of virtual registers. Both registers and word literals can serve as operands o for instructions. Programs P consist of lists of functions f with instructions i. We include instructions for moving values, performing arithmetic, accessing memory, and executing plain jumps. We also include while loops and conditional instructions. We do not include a call instruction, as we use plain jumps for function calls and manage the call stack explicitly. While we do not include reference counting in our formalization, we add an instruction **`unique`** to check whether a pointer is unique. This allows us to avoid copying continuations for single-shot resumptions.

Instructions:

$$
\begin{array}{lcl}
P & := & \overline{f : i} \\
i & := & r = \texttt{mov}\ o;\ i \\
 & | & r = \texttt{add}\ o\ o;\ i \\
 & | & r = \texttt{sub}\ o\ o;\ i \\
 & | & r = \texttt{alloc}\ o;\ i \\
 & | & r = \texttt{load}\ o[o];\ i \\
 & | & \texttt{store}\ o\ o[o];\ i \\
 & | & \texttt{jump}\ o \\
 & | & \texttt{while}\ r : i;\ i \\
 & | & \texttt{if}\ r : i;\ i \\
 & | & \texttt{unique}\ r;\ i
\end{array}
$$

Operands:

w	:=	1, 2, f, null, `@a5f`, ...	Words
r	:=	r_1, r_2, ...	Registers
o	:=	$w \mid r$	Operands

Syntax of Machine States:

M	::=	$\langle\, i,\ R,\ H,\ P \,\rangle$	Machine state
R	::=	$\{\ r \Rightarrow w\ \}$	Register state
H	::=	$\{\ w \Rightarrow B\ \}$	Heap state
B	::=	$\overline{w}$	Memory block

Stepping Relation:

$$
\begin{array}{lcll}
\langle\, r = \texttt{mov}\ o;\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto \mathcal{V}(o),\ R \mid H \,\rangle & \\
\langle\, r = \texttt{add}\ o_1\ o_2;\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto \mathcal{V}(o_1) + \mathcal{V}(o_2),\ R \mid H \,\rangle & \\
\langle\, r = \texttt{sub}\ o_1\ o_2;\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto \mathcal{V}(o_1) - \mathcal{V}(o_2),\ R \mid H \,\rangle & \\
\langle\, r = \texttt{alloc}\ o;\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto w,\ R \mid w \mapsto B,\ H \,\rangle & w \text{ fresh}, B = \text{null} \times \mathcal{V}(o) \\
\langle\, r = \texttt{load}\ o_1[o_2];\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto B(\mathcal{V}(o_2)),\ R \mid H \,\rangle & w = \mathcal{V}(o_1) \text{ and } B = H(w) \\
\langle\, \texttt{store}\ o_0\ o_1[o_2];\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid R \mid H[w \mapsto B[\mathcal{V}(o_2) \mapsto \mathcal{V}(o_0)]\ ] \,\rangle & w = \mathcal{V}(o_1) \text{ and } B = H(w) \\
\langle\, \texttt{jump}\ o \mid R \mid H \mid P \,\rangle & \to & \langle\, P(f) \mid R \mid H \mid P \,\rangle & f = \mathcal{V}(o) \\
\langle\, \texttt{while}\ r : i_0;\ i \mid R \mid H \,\rangle & \to & \langle\, i_0;\ \texttt{while}\ r : i_0;\ i \mid R \mid H \,\rangle & \text{if } R(r) = \text{true} \\
 & \to & \langle\, i \mid R \mid H \,\rangle & \text{otherwise} \\
\langle\, \texttt{if}\ r : i_0;\ i \mid R \mid H \,\rangle & \to & \langle\, i_0;\ i \mid R \mid H \,\rangle & \text{if } R(r) = \text{true} \\
 & \to & \langle\, i \mid R \mid H \,\rangle & \text{otherwise} \\
\langle\, r = \texttt{unique}\ r_0;\ i \mid R \mid H \,\rangle & \to & \langle\, i \mid r \mapsto \text{false},\ R \mid H \,\rangle & \text{if } R(r_0) \in H \cup R \setminus \{r_0\} \\
 & \to & \langle\, i \mid r \mapsto \text{true},\ R \mid H \,\rangle & \text{otherwise}
\end{array}
$$

where $\mathcal{V}(o) = R(o)$ if o is a register, and o otherwise

Fig. 9. Syntax of programs and instructions in ASM, and its semantics in terms of an abstract machine.

Semantics. Figure 9 also defines the small-step operational semantics of our target language ASM in terms of an abstract machine. The machine state consists of the current instruction i being executed, the register state R, the heap state H, and the program P. The register state maps registers to words, the heap state maps words to memory blocks, which are sequences of words, and the program maps functions to instructions. The semantics of the instructions is as expected. It uses an auxiliary function $\mathcal{V}(o)$ to trivially evaluate operands. We again omit the program P in most steps.

3.3 Compilation

Before giving a formal definition for the compilation of MCode to ASM, we provide a high-level intuition for the representation of values and machine configurations in MCode:

- **Current call stack.** We represent the current runtime stack K of the machine configuration by a pointer to a meta stack, which is a linked list of stacks, segmented at every delimiter **reset** m (also see Figure 1 again). This pointer is kept in a special register `msp`. In our implementation, each stack in this linked list is a dynamic array that grows exponentially, and we check for overflow whenever we push a frame or a reference. In this presentation, however, we assume that stacks always have enough space. Each stack contains a current memory offset, a pointer to a prompt, a link to the next element, and a flat list of frames.

$$
\begin{array}{lll}
[\![\overline{\textbf{def}\ f(\overline{x})\ =\ s}]\!] & ::= & \overline{f:\ \overline{x_i\ =\ \textbf{mov}\ a_i};\ [\![s]\!]} \\
[\![\textbf{jump}\ f(\overline{x})]\!] & ::= & \overline{a_i\ =\ \textbf{mov}\ x_i};\ \textbf{jump}\ f \\
[\![\textbf{push}(\overline{y})\ \{\ x \Rightarrow s_0\ \};\ s]\!] & ::= & \overline{\textbf{push}(y)};\ \textbf{push}(f);\ [\![s]\!] \\
 & & \text{where } f:\ x\ =\ \textbf{mov}\ a_1;\ \overline{y\ =\ \textbf{pop}()};\ [\![s_0]\!] \\
[\![\textbf{return}(y)]\!] & ::= & r\ =\ \textbf{pop}();\ a_1\ =\ \textbf{mov}\ y;\ \textbf{jump}\ r \\
[\![c\ =\ \textbf{new}(\overline{y})\ \{\ (\overline{x}) \Rightarrow s_0\ \};\ s]\!] & ::= & c\ =\ \textbf{alloc}\ (\mathtt{size}(\overline{y})\ +\ 1);\ \textbf{store}\ f\ c[0];\ \overline{\textbf{store}\ y_i\ c[i+1]};\ [\![s_0]\!] \\
 & & \text{where } f:\ \overline{y_i\ =\ \textbf{load}\ \mathsf{ep}[i+1]};\ \overline{x_i\ =\ \textbf{mov}\ a_i};\ [\![s]\!] \\
[\![\textbf{call}\ c(\overline{y})]\!] & ::= & r\ =\ \textbf{load}\ c[0];\ \mathsf{ep}\ =\ \textbf{mov}\ c;\ \overline{a_i\ =\ \textbf{mov}\ y_i};\ \textbf{jump}\ r \\
[\![r\ =\ \textbf{ref}\ y;\ s]\!] & ::= & r_o\ =\ \textbf{load}\ \mathtt{msp}[0];\ r_p\ =\ \textbf{load}\ \mathtt{msp}[1];\ \textbf{push}(y);\ \textbf{push}(\mathtt{popref});\ [\![s]\!] \\
[\![x\ =\ \textbf{get}\ r;\ s]\!] & ::= & k\ =\ \textbf{load}\ r_p[0];\ x\ =\ \textbf{load}\ k[r_o];\ [\![s]\!] \\
[\![\textbf{set}\ r\ y;\ s]\!] & ::= & k\ =\ \textbf{load}\ r_p[0];\ \textbf{store}\ y\ k[r_o];\ [\![s]\!] \\
[\![p\ =\ \textbf{reset};\ s]\!] & ::= & p\ =\ \textbf{reset}();\ [\![s]\!] \\
[\![k\ =\ \textbf{shift}\ p;\ s]\!] & ::= & k\ =\ \textbf{shift}(p);\ [\![s]\!] \\
[\![\textbf{resume}\ k;\ s]\!] & ::= & \textbf{if unique}(k):\ k'\ =\ \textbf{mov}\ k;\ \textbf{else}:\ k'\ =\ \textbf{copy}(k); \\
 & & \textbf{revalidate}(k');\ \textbf{resume}(k');\ [\![s]\!]
\end{array}
$$

Auxiliary Definitions:

```
push(x) :=
  offset = load msp[0]
  store x msp[offset]
  offset = add offset 1
  store offset msp[0]

x = pop() :=
  offset = load msp[0]
  offset = sub offset 1
  x = load msp[offset]
  store offset msp[0]

popref :
  x = pop();
  r = pop();
  jump r

underflow :
  p = load msp[1]
  store null p[0]
  msp = load msp[2];
  r = pop();
  jump r
```

Fig. 10. Compilation from MCode to ASM.

- **Frames.** Each stack frame is represented as a sequence of words corresponding to its environment and a function pointer f to return to. Reference frames are represented in the same way, where the environment contains the current state. The last frame of each stack is a special underflow frame with an empty environment, which returns to the next stack.
- **Closures.** We represent closures as a function pointer followed by an environment, which in turn is a sequence of words.
- **Continuations.** Continuations are represented as a cyclic linked list of stacks, and the handle for this object points to the last stack of the list. This way, we have fast access to both the first and last stacks of a continuation.
- **Markers.** Finally, we represent markers corresponding to prompts as pointers to a memory cell containing a pointer to a stack, and we represent markers corresponding to references as a pair of a prompt and an offset.

Figure 10 defines the compilation of statements in MCode to instructions in ASM. We compile function definitions $\textbf{def}\ f(\overline{x})\ =\ s$ to labels with the same name. They first move special argument registers into the parameter registers and execute the compiled body. Correspondingly, when we jump with arguments $\textbf{jump}\ f(\overline{x})$, we move them into the argument registers and perform a direct jump to the function label.

When we push a frame $\textbf{push}(\overline{y})\{\ x \Rightarrow s\ \}$, we first push each value of the environment and then push the return address f, which in turn moves the return value, pops the environment, and

executes the compiled rest of the statement. When we return, we pop the return address and jump to it with the return value. We define pushing and popping of words as auxiliary helper functions in Figure 10. Unlike a normal stack pointer, our meta stack pointer `msp` does not point directly to the next free location but to the base of the first element of the meta stack. For multiple consecutive pushes or pops, we avoid repeated additions, subtractions, loads, and stores.

For new closures $\textbf{new}(\overline{y})\{\ (\overline{x}) \Rightarrow s\ \}$, we allocate space, store a function pointer f, and store the environment as a sequence of words. The function pointer restores this environment from a special `ep` register and moves the arguments into parameter registers. When we invoke a closure, we load the function pointer, move the closure into the `ep` register, move the arguments into their registers, and perform an indirect jump to the loaded function pointer.

We represent references with two registers, r_p and r_o, for a prompt and an offset, respectively. We allocate a reference $r = \textbf{ref}\ y$ by recording the stack offset and the current prompt, then pushing the value y and a function pointer popref that will pop the state upon return.

To get and set the value of a reference, $x = \textbf{get}\ r$ and $\textbf{set}\ r\ y$, we load the current stack k at the prompt in r_p. We then load respectively store the value on this stack at the offset r_o. Indeed, getting and setting references is constant time, both taking only two instructions. The additional indirection is required because we support stack-allocated mutable state in presence of multi-shot continuations. It also allows us to grow stacks by reallocation.

The compilation of delimited control operators is more intricate. We discuss the auxiliary definitions of **`reset`**, **`shift`**, **`resume`**, and **`revalidate`** in the next subsection.

3.4 Delimited Control Operators

Each stack on the meta stack has an associated prompt, representing a marker in MCode. Prompts stay at a stable address and contain only a single pointer to a stack. We call a stack *valid* if its prompt points back to the stack itself. We call a stack *invalid* if the prompt points to either a different stack or null. The meta stack *only* contains valid stacks. This way, we have direct access to a delimiter without any traversal of the meta stack.

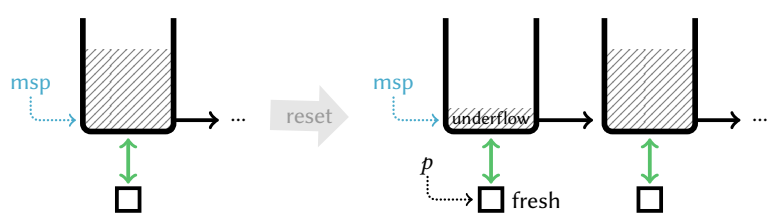
When we resume a continuation, we need to know whether the contained stacks are valid. For this, we maintain the following invariant: a continuation may only contain *any* valid stacks if *all* of them are valid. This allows us to determine if all stacks in a continuation are valid in constant time. Moreover, since captured continuations are valid, we achieve constant-time capture and resumption for one-shot use of continuations.

For multi-shot continuations, we need to copy the stacks. This process creates *invalid* stacks. If we want to resume an *invalid* continuation, we need to revalidate it. We traverse the continuation, revalidating each stack by updating its prompt. The prompt might already point to a valid stack, in which case we must invalidate it first. The invalidation process might start in the middle of a continuation, which is why we represent continuations as cyclic linked lists.

We now describe the compilation of delimited control operators in more detail.

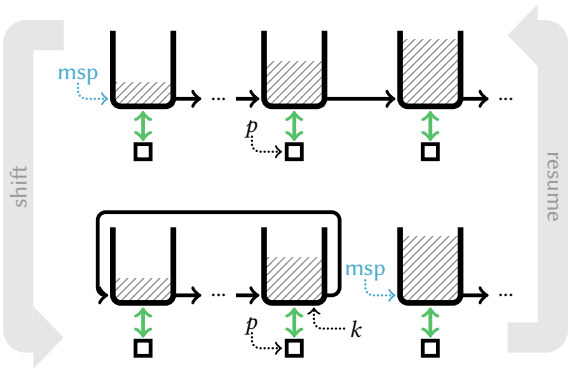
Reset. Figure 11 illustrates the situation before and after a reset. The statement $p = \textbf{reset}$ pushes a delimiter with a fresh marker m onto the stack K. The **`reset`** operation allocates a new prompt p and a new stack k, which point to each other, and pushes the stack onto the meta stack. In the stack, we initialize its offset to 4, its associated prompt to p, its next element to `msp`, and push an underflow function pointer that will load and return to the next stack. We then store the stack in prompt p and move it into the meta stack pointer `msp`. The underflow frame stores `null` in the prompt, loads the next stack, and returns to it.

Shift & Resume. Figure 12 illustrates both shift and resume, as the two are inverses of each other. Whereas the statement $k = \textbf{shift}\ m$ captures the part of the meta stack up to delimiter m as




```
p = reset() :=
  p = alloc 1
  k = alloc enough
  store 4 k[0]
  store p k[1]
  store msp k[2]
  store underflow k[3]
  store k p[0]
  msp = mov k
```

Fig. 11. Reset creates a fresh stack and a fresh prompt, links them, and pushes them onto the meta stack.




```
k = shift(p) :=
  k = load p[0]
  k′ = load k[2]
  store msp k[2]
  msp = mov k′

resume(k) :=
  k′ = load k[2]
  store msp k[2]
  msp = mov k′
```

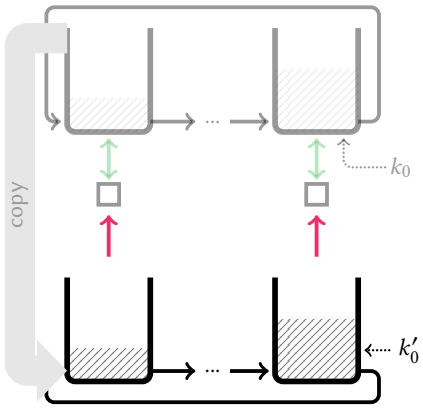
Fig. 12. Shift and resume only amounts to switching two pointers.

a continuation k, the statement **resume** k pushes a continuation k back onto the meta stack. Operation **shift** captures the current continuation by cutting the meta stack at prompt p, sets the meta stack pointer to the next stack k', and returns the captured continuation k. The **resume** operation assumes that k is a unique reference to a valid continuation and pushes it back onto the meta stack. It first updates the meta stack pointer to the first stack k' in the continuation and then points the last stack k to the previous meta stack pointer.

Copying Continuations. The code presented so far works, as long as every continuation is resumed at most once. However, we support (non-reentrant) multi-shot continuations and local mutable references, which we explain next. In many programs, we are on the fast path presented so far, after a quick check for unique ownership and validity. When we attempt to resume a continuation that we do not uniquely own, we create a copy that we do uniquely own.

Figure 13 depicts the situation before and after creating a copy k_0' of continuation k_0. We copy the memory of each stack in k_0. We traverse the linked list until we find the end, which is marked by pointing back to the head k_0. We also mark the end of k_0' to point back to its head. The copy is invalid. Since we very often resume the copy afterwards, as a micro-optimization, we could revalidate it (explained in the next subsection) while creating the copy.

Revalidate & Invalidate. Before we resume a continuation, we must check its validity and potentially revalidate it. We maintain the invariant that if an element of a continuation is valid, then the rest is valid as well. This means that for valid continuations, we only need to check the first element for validity and can resume immediately without traversing the continuation. However, if the stacks




```
k0' = copy(k0) :=
  k0' = alloc enough
  size = load k0[0]
  memcpy k0' k0 size
  s' = mov k0'
  k = load k0[2]
  while (k ≠ k0) :
    k' = alloc enough
    size = load k[0]
    memcpy k' k size
    store k' s'[2]
    s' = mov k'
    k = load k[2]
  store k0' k'[2]
```

Fig. 13. Copying a continuation k (greyed out, since it remains unchanged) results in a deep copy of each stack with *invalid* prompt pointers. That is, the prompt does not point back to the copy of the stack.

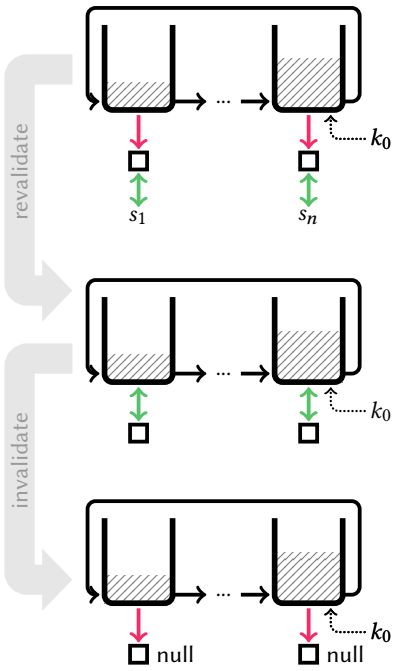
of the continuation are invalid, we traverse the continuation and, for each stack, point its prompt back to that stack. Figure 14 depicts the situation before and after revalidation. To maintain the aforementioned invariant, we also invalidate the continuation it previously pointed to. Figure 14 also depicts the situation before and after invalidation. To invalidate a continuation, we traverse all elements and make their associated prompts point to null, a special pointer that is not equal to any other one. This way, any validity check for any element will fail, and we will eventually have to revalidate the continuation, setting the prompts to point back to these stacks.

Time Complexity. In cases where continuations are unique and resumed only once, **shift** and **resume** are constant-time operations. They do not traverse the continuation and execute only a handful of instructions. In this case, stacks always remain valid, as invalid stacks are only introduced by copying continuations or resuming copies, which only occurs if they are used multiple times.

When continuations are resumed multiple times, our implementation introduces an overhead linear in the number of captured frames, since we must create a copy of each stack in the continuation. We believe that this overhead is unavoidable to ensure the correct semantics of multi-shot delimited control in the presence of stack-allocated mutable references. The revalidation of continuations is linear[1] in the number of stacks in the continuation but is only necessary when invalid stacks are present, in which case there is already a linear overhead from copying continuations.

Due to our requirement of validity, we can access any given marker on the meta stack in constant time, which is important for the implementation of local mutable references. We can allocate state directly on the stack and have constant-time access to it while maintaining the correct backtracking behavior in the presence of multiple resumptions. Whenever we copy a continuation, any mutable state that it captured is naturally copied as well. There is still some overhead from the indirection through the prompt and the offset. However, with direct pointers to mutable state, we would lose position independence of stacks, which would make resuming from a handler more costly.

[1]The revalidation of continuations with n stacks might invalidate n other continuations, but since this is a delayed invalidation, we still have linear amortized time complexity.




```
revalidate(k0) :=
  k = mov k0
  p = load k[1]
  s = load p[0]
  while (k ≠ s):
    if (s ≠ null):
      invalidate(s)
    store k p[0]
    k = load k[2]
    p = load k[1]
    s = load p[0]

invalidate(k0) :=
  k = mov k0
  p = load k[1]
  s = load p[0]
  while (k == s):
    store null p[0]
    k = load k[2]
    p = load k[1]
    s = load p[0]
```

Fig. 14. To revalidate a continuation, we traverse all prompts and point them to the respective stack. Since the prompts potentially pointed to another stack s_i, we additionally need to invalidate it before. Invalidation traverses a continuation and sets the prompts to null.

4 Implementation

To evaluate the practical feasibility of our compilation presented in Section 3, we implemented it in the existing compiler for Effekt [Brachthäuser et al. 2020]. In addition to lexical effect handlers, the language Effekt features algebraic data types, interface types, a standard library with immutable and mutable data structures, common effects and handlers, several medium-sized case studies, and an extensive test suite. The compiler supports multiple backends for the same source language, with a shared intermediate representation. It performs various high-level optimizations on this shared intermediate representation, such as eliminating continuation capture in tail-resumptive handlers. From the shared intermediate representation, we generate code in an intermediate language similar to MCode, which we then compile to LLVM intermediate code, closely following the compilation to ASM described in Section 3.3. We have a runtime system that we wrote directly in LLVM's intermediate representation, containing functions to manage the meta stack. The runtime system code is compiled by LLVM together with the generated code to enable further optimizations.

4.1 Differences to the Formalization

Our presentation of MCode in Section 3 is for a simplified subset of the intermediate representation that we have actually implemented. Minor differences are the addition of algebraic data types and the generalization of closures to objects with multiple methods. We now explain larger differences.

Reference Counting. Our runtime system uses *garbage-free reference counting* [Reinking et al. 2021] to manage memory. Each heap-allocated value has a reference counter at its base. Whenever a

value is used multiple times within a scope, we share it by incrementing its reference count. We erase a reference when it is *used* for the last time in a scope, rather than at the end of a scope, by decrementing its reference count and freeing it when it drops to zero. This ensures that every value is freed as soon as it is no longer needed. We pass around capabilities that, in turn, close over prompts and references to access parts of the meta stack. Due to the effect safety of our source language, we can assume that prompts outlive any capabilities and references that use them. Therefore, they only have weak references to prompts, and sharing them does not increase their reference count, nor does erasing them decrease it. We only count the number of stacks that reference a prompt. Stack frames have associated sharer and eraser functions, which are called when a stack is copied or freed. These functions are responsible for incrementing and decrementing the reference counters of the values in the stack frames. Therefore, in our implementation, exceptions must traverse the stack to free it, taking time linear in the number of frames. Heap objects, specifically closures, also have an associated eraser function, which is called when a closure is freed to traverse and erase its environment.

Stack Growth. In our implementation, whenever we push something onto a stack, we check if there is enough space left. If not, we reallocate the stack. This is a linear-time operation, but we amortize it to be constant-time by doubling the size of the stack each time. This is, in addition to copying continuations, another reason why references to stacks need the indirection through prompts.

Regions. Effekt features regions into which we dynamically allocate mutable references. Regions introduce a new scope, and memory allocated within a region is freed when the region is exited. This is useful for creating local mutable references that may outlive the current scope. Our motivating example from Section 2 (left) can be equivalently expressed in the following way (middle).

Mutable variables

```
var i = 0;
checkpointing {
  var s = 0;
  ...
}
```

Regions in Effekt

```
region outer {
  checkpointing {
    region inner {
      var i in outer = 0;
      var s in inner = 0;
      ... } } }
```

Regions in MCode

```
allocate(r, v) :=
  k = shift r;
  x = ref v;
  resume k;
  return x
```

The `region r { ... }` construct binds the name `r` for a freshly created region in the delimited body. Inside, we allocate a mutable reference `x` in region `r` with initial value `v` using `var x in r = v`. Neither the regions nor the references shall escape, and the type system of Effekt ensures this. Regions do not appear in MCode but are desugared to a combination of prompts and references. The `region` statement desugars to a `reset`, and the region `r` is the freshly created prompt. To allocate a value `v` into region `r`, we define the following code above (right column). Getting and setting the value of a reference allocated in a region is the same as with ordinary references and takes constant time, regardless of how far away the region is on the meta stack.

Bidirectional handlers. Effekt features bidirectional handlers [Zhang et al. 2020], which allow for resumption with a computation instead of a value. For example, we can declare that the `await` operation might raise a `retry` exception when used. The handler for `await` now can resume by raising `retry` at the use-site.

```
effect await(): Result / retry        resume { do retry() }      resume k; call retry()
```

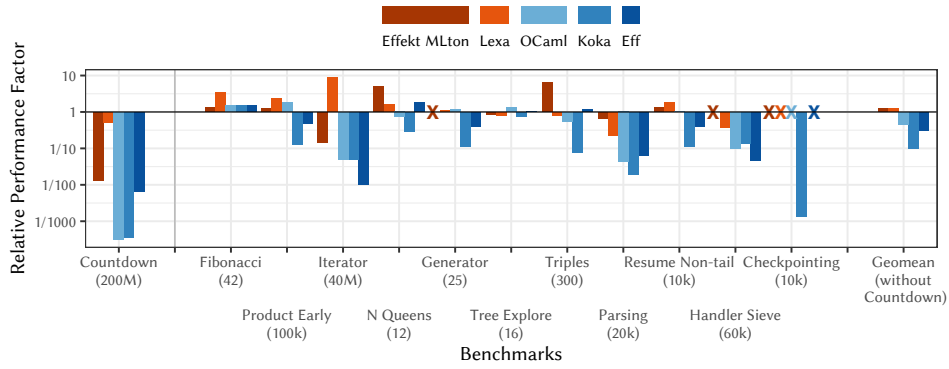


Fig. 15. Performance of Effekt compared with other implementations, normalized to Effekt. Bars that go down show slower performance than Effekt. Bars that go up show faster performance than Effekt. The y-axis is in log scale. Benchmarks that did not run are marked with an X.

The implementation of this feature is a natural desugaring to capability passing and resuming with a statement. Effekt supports a variation of scoped effects in a similar way.

Asynchronous input and output. Effekt features lightweight cooperative concurrent tasks and asynchronous input and output using the libuv [2025] library. Input and output operations are written in direct style with no additional annotation. For example, we can spawn a task that runs concurrently and sleeps without blocking the main thread.

```
spawn(box { println("task: hello"); sleep(2); println("task: world") });
println("program: hello"); sleep(1); println("program: world")
```

Spawning a new task amounts to allocating a new tiny meta stack on which it runs. Stacks grow as required. Suspending the current computation amounts to an ordinary function call, since the meta stack contains all information we need to resume. The type system of Effekt prevents effects, including exceptions and mutable references, from affecting the context outside of the current task.

5 Performance Evaluation

To evaluate the performance of our implementation, we conducted a number of experiments. Foremost, we are interested in the performance cost of supporting multiple resumptions and local mutable state. Since we generalize the approach of Ma et al. [2024], we are particularly interested in determining whether we can achieve the same high levels of performance while implementing a more general language that can express a broader range of programs. We also compare our approach to other implementations of both lexical and dynamic effect handlers. Finally, we aim to confirm that our implementation resolves the undesired non-linear scaling properties of deeply nested effect handlers in previous implementations of Effekt, as identified by Ma et al. [2024].

Performance Comparison with Lexical and Dynamic Effect Handler Implementations. We compare our new Effekt implementation (version 0.22.0) against five other implementations: Effekt MLton (version 0.2.2) is a now-discontinued MLton backend that compiled effect handlers to iterated continuation-passing style [Müller et al. 2023]. This backend did not support the full set of Effekt's features and could not compile all benchmarks. Together with Lexa [Ma et al. 2024], both implement lexical effect handlers. Moreover, we compare with OCaml (version 5.1.1) [Sivaramakrishnan et al. 2021], Koka (version 3.1.2) [Leijen 2017b], and Eff (version 5.1) [Plotkin and Pretnar 2013], which all

	Indirect Ref.	Direct Ref.
Without Opt.	110.3 (± 1.9) ms	88.4 (± 3.2) ms
With Opt.	43.6 (± 1.1) ms	21.8 (± 0.9) ms

```
main :
  x = ref 100000000;
  push i ⇒ () ;
  jump loop(x)
```

```
loop(x) :
  i = get x;
  if (i == 0) :
    return i
  else :
    set x (i − 1);
    jump loop(x)
```

Fig. 16. Runtime of accessing mutable state in a tight loop. We compare indirect references (as presented in this paper) with using direct references in this benchmark, both with and without LLVM optimizations.

implement dynamic effect handlers. We use a community maintained benchmark suite designed specifically for languages with effect handlers, covering a wide range of use cases [Hillerström et al. 2023]. In addition, we include the Checkpointing example from Section 2. It requires support for multiple resumptions and local mutable state, which Lexa and OCaml lack. Unfortunately, while Eff would support this example, its compiler crashes with an internal error. We measured the benchmarks on a machine with a 16-core AMD Ryzen 7 PRO 7840U processor and 64 GB of RAM. We performed each experiment at least 10 times and report median runtimes. We submit the benchmarked code and our raw results as supplementary material.

Figure 15 shows the performance of each implementation normalized to our new Effekt backend. Bars that go down show worse performance compared to Effekt. Bars that go up show a better performance. Benchmarks that could not be run are marked with a colored X. The figure shows that Effekt indeed performs well compared to the other implementations, particularly the ones implementing dynamic effect handlers. The geometric mean on the right, shows that we perform significantly better than OCaml (2.2x faster), Eff (3.2x), and Koka (10.2x). The benchmarks that use multiple resumptions in OCaml use a third-party library to manually clone continuations. The benchmarks that use mutable state in Eff do so purely with effect handlers and do not use mutable references. Our implementation shows competitive performance to the prior backend Effekt MLton and Lexa which are about 1.25x and 1.29x faster. Lexa is not able to run the Checkpointing benchmark as it requires multiple resumptions in a handler that is not the closest one. Our implementation enables some aggressive optimizations that sometimes provide significant benefits. For example, the Countdown benchmark is largely evaluated at compile time and the Iterator, Parsing, and Handler Sieve benchmarks benefit from Effekt's tail resumption optimization and from the direct fast access to local mutable references. As the optimizations for Countdown are a clear outlier, we have excluded this benchmark from the geometric mean reported above. We do not perform well on Product Early, as for exceptions we have to erase the continuation in linear time. Lexa, performs significantly better on the Iterator benchmark where we believe to pay an overhead due to our prompt indirection. For Fibonacci we believe the reason for the worse performance is our custom calling convention and stack management which LLVM is not able to optimize.

Overhead of the indirection in local mutable references. We access local mutable references through an indirection via a prompt and an offset. This introduces overhead compared to having a direct pointer to the state. We chose this implementation strategy to support multiple resumptions. To evaluate the overhead, we modified our runtime system to use direct pointers instead. This implementation does not support multiple resumptions anymore. Therefore, we do not run benchmarks with multi-shot handlers, and increase the initial stack size, so it does not need to be reallocated. When comparing this modified implementation to the presented implementation, we did not observe a significant overhead with performance differences below 2%. We inspected benchmarks

which make use of local state, such as Countdown, and observed that in both versions the optimizer removes most memory operations at compile time. When we turned optimizations off, we observed that the overall runtime was much higher and that both versions showed still almost identical performance, as now the memory operations are insignificant compared to the rest of the program.

To get an estimate for an upper bound on the overhead, we manually translated the example of Figure 16 in MCode to LLVM IR, and measured its performance with and without the indirection for local mutable references. We compiled the program with and without optimizations, and marked the relevant state accesses with `volatile` to prevent them from being optimized away. Figure 16 shows the result for these programs. The overhead of the indirection is about 1.2x for unoptimized and about 1.9x for optimized code. An interesting observation is that the absolute difference between the two implementations is the same for both and reproducible with varying iteration counts. As expected, the worst case slowdown for the additional indirection is roughly a factor of two.

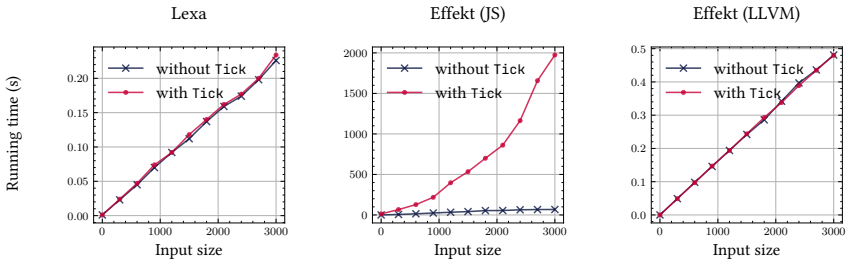


Fig. 17. Scaling behavior of deeply nested effect handlers in the prior JavaScript and new LLVM backends. The benchmarks install an unbounded number of nested handlers and the handler for the Tick effect is below all of them. (The MLton backend could not compile this benchmark due to effect-polymorphic recursion.)

Scaling of nested effect handlers. Ma et al. [2024] reported a non-linear scaling behavior in Effekt's JavaScript backend when using nested effect handlers due to the search for the handler. Figure 17 confirms that our new implementation fixes this undesired scaling behavior.

6 Limitation: Continuations are not Reentrant

While the approach presented in this paper supports resuming the continuation multiple times *sequentially*, we do not support resuming the same continuation *concurrently*. That is, our continuations are *not reentrant*. This is illustrated by the following examples.

Supported (Effekt)

```
resume(());
resume(())
```

Supported (MCode)

```
push(k) { () ⇒
  resume k;
  return () }
resume k;
return ()
```

Not Supported (Effekt)

```
resume {
  resume { () }
}
```

Not Supported (MCode)

```
resume k;
resume k;
return ()
```

The left two columns show examples where we resume a second time after finishing resuming for the first time. The right two columns show the situation where we try to resume, while another copy of the continuation is still on the meta stack. To admit constant-time lookup, prompts always point back to at most one active copy of the stack—having two copies on the stack simultaneously would violate this invariant. In fact, this is a limitation of MCode. Reentrant continuations lead to repeated markers on the stack, which exhibit undefined behavior for the shift, get and set operations. In our implementation, we detect this situation at runtime and abort the program.

While it is very easy to construct a situation like above in MCode, reproducing it in a source language requires at least one of the following features:

First-class functions. With first-class functions (added to Effekt by Brachthäuser et al. [2022]), it is possible to resume with a thunk that will resume again, as illustrated in the following example.

```
effect thunk(): () ⇒ Unit at {}

try { do thunk()() }
with thunk {
resume(box { resume(box { () }) }) }
```

Bidirectional effects. A very similar situation can be achieved by means of bidirectional effects [Zhang et al. 2020], that is, when effect operations can use effects on their own.

```
effect yield(): Unit
effect thunk(): Unit / yield

try { do thunk() }
with thunk { resume { resume { do yield() } } }
with yield { resume(()) }
```

Parallelism. If a language supports parallel execution, which Effekt currently does not, it would be possible to resume twice at the same time, as in the following example.

```
effect fork(): Bool

try { println(do fork()) }
with fork {
  spawn(box { resume(true) })
  resume(false) }
```

The meta stack is a linked list ending in a null pointer, while the pointers in captured continuations form cycles. Before resuming a shared continuation, we create a copy and invalidate the currently valid copy. On a reentrant resumption, that copy is already part of the meta stack. This means that we encounter a null pointer during invalidation, which we detect and abort the program.

In general, it might be possible to devise a type system to statically prevent this situation. This would amount to tracking the resumption on the type-level in order to prevent resuming it in a scope where it is already being resumed. In Effekt, resumptions are *transparent*, this means the type checker annotates them with the capabilities they close over. In the first example of this section, this means the continuation is indistinguishable from a pure function of type `() ⇒ Unit at {}`. This makes it difficult to statically detect reentry without loss of expressivity.

All Effekt backends, prior to our implementation, supported reentrant resumptions. However, using our approach to execute about 25.000 lines of Effekt code, heavily using effects and also multiple resumptions, we encountered not a single reentrancy error. A reason could be that first-class functions and bidirectional effects have been retroactively added to the language and are thus used less frequently. While existing programs do not seem to rely on reentrancy, there are, of course, valid use cases of reentrant resumptions, which at the moment lead to a runtime error. These runtime errors might be particularly suprising for users of a library, who are not aware that the library is implemented in terms of effect handlers. In the future, it thus appears important to investigate runtime support for reentrancy or rule it out statically by means of a type system.

7 Related Work

We presented a compiler and runtime system for lexical effect handlers that support multi-shot continuations and stack-allocated state. We express lexical effect handlers in terms of delimited control operators for capturing the continuation up to a prompt and resuming it later [Gunter

et al. 1995] In this section, we compare with other runtime systems supporting classical control operators or effect handlers. Finally, we discuss local mutable references and reallocation of stacks.

7.1 Runtime Support for Control Operators

Undelimited Continuations. Hieb et al. [1990] present an efficient implementation of `call/cc`, a control operator that captures the undelimited continuation. They introduce segmented stacks, a technique to enable a dynamically growing call stack without moving too much data. While our meta stack might look like a segmented stack, its purpose is different as it enables fast capturing of delimited continuations. Our stack growth strategy is an orthogonal decision: we reallocate stacks. Bruggeman et al. [1996] observe that many continuations are invoked only once and extend a runtime system based on segmented stacks with special support for undelimited one-shot continuations, which avoid the copying overhead of multi-shot continuations. They offer a control operator `call/1cc` to the programmer that captures a one-shot continuation, and it is an error to resume it more than once. In contrast, we inspect the reference count at runtime to detect if we are resuming for the last time and avoid copying in this case. Clinger et al. [1999] present a comprehensive survey and comparison of different implementation techniques for undelimited continuations. They measure instruction counts and discuss indirect, non-measurable costs. Similarly, more recently, Farvardin and Reppy [2020] present a comprehensive survey and comparison of different implementation techniques for one-shot continuations that have stack extent with the control operator `call/ec`. Both conclude with a list of trade-offs one must consider, which we took into account during development.

Delimited Continuations. Gasbichler and Sperber [2002] present a direct implementation of delimited control operators `reset` and `shift` for Scheme, with improved performance over the encoding via undelimited control. When capturing the continuation, they search for the closest delimiter, which takes time linear in the number of frames. They support multiple resumptions by copying incrementally between the heap and the stack. Masuko and Asai [2009] present a compiler and runtime system for delimited control operators `reset` and `shift` for the MinCaml language. They present one implementation that eagerly copies frames upon continuation capture, which takes time linear in the number of frames, and one implementation that does so lazily. This variant still takes time linear in the number of frames upon resumption, but is optimized to take constant time in case of tail resumption. They support multiple resumptions but do not discuss stack-allocated state. Kiselyov [2012] presents an implementation of multi-prompt delimited control for the OCaml bytecode interpreter as a library using low-level instructions. He uses exception handler frames as markers and copies parts of the stack when a continuation is captured, which takes time linear in the number of frames. He supports multiple resumptions but does not discuss stack-allocated state. Pham and Odersky [2024] extend the compiler and runtime system of Scala Native with operators for multi-prompt delimited control. They keep a linked list of active handlers in a thread-local variable. They move frames out of the stack into a continuation object and copy them back upon resumption, which takes time linear in the number of frames. They choose not to support multiple resumptions to avoid violating resource constraints.

7.2 Runtime Support for Effect Handlers

Dynamic Effect Handlers. Hillerström et al. [2017] present a continuation-passing translation for dynamic effect handlers. At runtime, they pass a linked list of pairs of continuations and handlers. When performing an effect operation, they search this linked list for a matching handler, accumulating a linked list in reverse order, which they replay upon resumption. All these operations take time linear in the number of handlers. They fully support multiple resumptions. Xie and Leijen [2021] implement dynamic effect handlers using generalized evidence passing and runtime support

for multi-prompt delimited control. When invoking an effect operation, they look up the handler implementation in an evidence vector that is passed to each function. This allows tail-resumptive operations to avoid capturing the continuation, but due to the nature of dynamic effect handlers, it requires special frames to correctly resolve effect operations in those cases. Capturing continuations searches for a matching prompt and is linear in the number of handlers. They fully support multiple resumptions. Sivaramakrishnan et al. [2021] present the design and implementation of dynamic effect handlers for OCaml, an industrial-strength functional programming language, maintaining backwards compatibility and the performance of existing programs. Like us, they represent the meta stack as a linked list of stacks and grow stacks by reallocation. Searching for a matching handler is linear in the number of handlers. They make the pragmatic choice of supporting only one-shot continuations. A third-party library implements cloning of continuations but warns about surprising behavior due to the compiler's heap-to-stack conversion of mutable references. Phipps-Costin et al. [2023] extend WebAssembly, a low-level portable code format, with dynamic effect handlers. They provide a formal specification, an extension to the reference interpreter, and a prototype extension of a production-grade interpreter. They too make the pragmatic choice to support only one-shot continuations. Like us, they keep a linked list of stacks that supports lightweight stack switching. Searching for a matching handler is linear in the number of handlers. Interestingly, they have specified an additional **resume_throw** instruction to resume with an exception. Both resuming with a value and resuming with an exception are subsumed by our ability to resume with an arbitrary computation. Alvarez-Picallo et al. [2024] present a C library for dynamic effect handlers based on mutable coroutines. Naturally, this implementation only permits one-shot continuations, again a pragmatic choice. They keep a linked list of active coroutines, analogous to our meta stack. Instead of reallocation, they use either segmented stacks or fixed-size stacks. Searching for a matching handler is linear in the number of handlers.

Lexical Effect Handlers. Brachthäuser et al. [2018] present a Java library for lexical effect handlers in capability-passing style. They build upon multi-prompt delimited control and, following Dybvig et al. [2007], represent the meta stack as a linked list of stacks, where each stack is a linked list of frames. They instrument JVM bytecode to update this meta stack when pushing a frame. Interestingly, they too provide a method `unwindWith` to resume with an exception, which is subsumed by our ability to resume with an arbitrary computation. Because they employ explicit capability passing, handler implementations are immediately available when an effect operation is used. However, capturing continuations takes time linear in the number of handlers. Ghica et al. [2022] present a C++ library for lexical effect handlers, using an existing library for suspending and resuming runtime stacks. They only support one-shot continuations, again a pragmatic choice. They generate a fresh label for each handler instance at runtime, implementing lexical handlers. Their meta stack is a linked list of stacks, each corresponding to a handler and search for a matching handler, takes time linear in the number of handlers. Ma et al. [2024] present a language with lexical effect handlers and constant-time continuation capture and resumption. Their language, Lexa, is compiled to C and uses the C stack and the System V calling convention. To implement effect handlers, they use a library with inline assembly for switching between stacks. Similar to our approach, they use a linked list of stacks to support delimited control operators. In Lexa, effect handlers are stored at the stack where they are introduced, and effectful functions receive a direct pointer to that stack as an additional parameter. Since Lexa uses direct pointers to stacks, stacks remain position dependent, which prohibits multi-shot resumptions from capturing any additional handler. Our work is based on theirs, but we support non-reentrant multi-shot continuations in the presence of stack-allocated state while keeping constant-time capture and resume for the one-shot case.

7.3 Runtime Support for Local Mutable References

The Go programming language features lightweight threads, represented as separate stacks. These are allocated with a small initial size and grow on demand. Starting with version 1.3, the implementation changed from using segmented stacks to using reallocation[2]. Growing stacks by reallocation requires adjusting all pointers into the old stack to point into the new stack, which the Go runtime does for pointers from the stack into the stack. It prevents pointers from the heap into the stack by allocating the memory for those references that might escape on the heap. Likewise, the runtime system of OCaml [Sivaramakrishnan et al. 2021] grows stacks by reallocation. Its runtime system keeps a chain of exception handlers that are allocated on the stack. These pointers are traversed and adjusted upon reallocation. Moreover, the compiler sometimes performs an optimization to allocate mutable references on the stack instead of on the heap, which is only valid under the assumption that resumptions are one-shot [de Vilhena and Pottier 2021]. This assumption is only violated by third-party libraries for multi-shot continuations. The Effekt programming language [Brachthäuser et al. 2020] originally was introduced with second-class functions only: they can be passed but not returned nor stored. This restriction would enable an implementation where all handlers and all functions are allocated on the stack, which in turn would enable a runtime system that supports multi-shot resumptions and local mutable references by adjusting pointers upon reallocation. This implementation technique would in turn avoid the indirection for local mutable references while also supporting reentrant resumptions. A practical exploration of this alternative implementation technique is interesting future work. The current version of Effekt, however, does feature first-class functions [Brachthäuser et al. 2022]. These require the allocation of closures on the heap and consequently the existence of pointers from the heap into the stack. Other implementations of Effekt use a translation to iterated continuation-passing style [Müller et al. 2023], translating local mutable references to state passing, or use a runtime system which saves all local mutable references upon continuation capture and restores them upon resumption. Runtime systems of the Scheme programming language support continuation marks [Flatt et al. 2019, 2007]. They conceptually are dynamically bound maps from keys to values. The implementation in Chez Scheme [Flatt and Dybvig 2020] uses a dedicated mark register for a linked list of continuation marks. Access to these takes amortized constant time by caching the values of recently-used keys. Resuming a continuation creates a copy of it, which the runtime system sometimes avoids by detecting one-shot usage.

8 Conclusion

We presented a compiler and runtime system for lexical effect handlers that supports multi-shot continuations and local mutable references. Our approach builds on the work of Ma et al. [2024] and extends it to support multiple resumptions and additionally supports local mutable references. We achieve this by using garbage-free reference counting and a carefully designed runtime system that allows for constant-time continuation capture and resumption in the common one-shot case. Our evaluation demonstrates that our approach is feasible and that the implementation performs competitively with state-of-the-art systems for effect handlers, and in many cases, outperforms them. Moreover, our approach eliminates the undesired scaling properties of deeply nested effect handlers that were demonstrated by Ma et al. [2024]. Our results refute the common belief that multiple resumptions are incompatible with efficient stack-switching implementations, showing that efficiency does not need to come at the cost of expressivity.

[2]https://go.dev/doc/go1.3#stacks

Acknowledgments

We would like to thank the reviewers for their constructive feedback, which has significantly helped us improve our work. We also gratefully acknowledge Jonathan Frech, Marvin Borner, and Mattis Böckle for their contributions to the LLVM backend implementation of Effekt in their roles as student compiler engineers. The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

Mario Alvarez-Picallo, Teodoro Freund, Dan R. Ghica, and Sam Lindley. 2024. Effect Handlers for C via Coroutines. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 358 (Oct. 2024), 28 pages. doi:10.1145/3689798

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (Jan. 2019), 28 pages.

Jonathan Immanuel Brachthäuser. 2024. Optimize tailresumptive handlers. https://github.com/effekt-lang/effekt/pull/674. PR #674.

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. *Proc. ACM Program. Lang.* 6, OOPSLA, Article 76 (apr 2022), 30 pages. doi:10.1145/3527320

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 111 (Oct. 2018), 27 pages. doi:10.1145/3276481

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). doi:10.1145/3428194

Jonathan Immanuel Brachthäuser and Daan Leijen. 2019. *Programming with Implicit Values, Functions, and Control.* Technical Report MSR-TR-2019-7. Microsoft Research.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 99–107. doi:10.1145/231379.231395

William D Clinger, Anne H Hartheimer, and Eric M Ost. 1999. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation* 12 (1999), 7–45.

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

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. 2007. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17, 6 (Nov. 2007), 687–730. doi:10.1017/S0956796807006259

Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 75–90. doi:10.1145/3385412.3385994

Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. 2019. Rebuilding Racket on Chez Scheme (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 78 (July 2019), 15 pages. doi:10.1145/3341642

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 45–58. doi:10.1145/3385412.3385981

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *Proceedings of the International Conference on Functional Programming* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/1291151.1291178

Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. 1984. Programming with Continuations. In *Program Transformation and Programming Environments*, Peter Pepper (Ed.). Springer-Verlag, Berlin, Heidelberg.

Martin Gasbichler and Michael Sperber. 2002. Final Shift for Call/Cc: Direct Implementation of Shift and Reset. In *Proceedings of the International Conference on Functional Programming* (Pittsburgh, PA, USA). ACM, New York, NY, USA, 271–282.

Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (Oct. 2022), 29 pages. doi:10.1145/3563445

Oliver Goldstein and Ohad Kammar. 2024. Modular probabilistic programming with algebraic effects (MSc Thesis 2019). arXiv:2412.19826 [cs.PL] https://arxiv.org/abs/2412.19826

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California,

USA). ACM, New York, NY, USA, 12–23.
Christopher T Haynes. 1987. Logic continuations. *The Journal of Logic Programming* 4, 2 (1987), 157–176.
Robert Hieb and R. Kent Dybvig. 1990. Continuations and Concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (Seattle, Washington, USA) *(PPOPP '90)*. ACM, New York, NY, USA, 128–136.
Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990. Representing Control in the Presence of First-class Continuations. In *Proceedings of the Conference on Programming Language Design and Implementation* (White Plains, New York, USA). ACM, New York, NY, USA, 66–77.
Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Formal Structures for Computation and Deduction (LIPIcs, Vol. 84)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
Daniel Hillerström, Filip Koprivec, and Philipp Schuster (benchmarking chairs). 2023. Effect handlers benchmarks suite. (2023). https://github.com/effect-handlers/effect-handlers-bench
Alexis King. 2022. Native, first-class, delimited continuations. https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7942. MR #7942.
Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theoretical Computer Science* 435 (2012), 56–76.
Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded Probabilistic Programming. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 360–384.
Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the International Conference on Functional Programming* (Portland, Oregon, USA). ACM, New York, NY, USA, 26–37.
Eric Koskinen and Maurice Herlihy. 2008. Checkpoints and continuations instead of nested transactions. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany) *(SPAA '08)*. Association for Computing Machinery, New York, NY, USA, 160–168. doi:10.1145/1378533.1378563
Daan Leijen. 2016. *Algebraic Effects for Functional Programming*. Technical Report. MSR-TR-2016-29. Microsoft Research technical report.
Daan Leijen. 2017a. Implementing Algebraic Effects in C. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer International Publishing, Cham, Switzerland, 339–363.
Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499. doi:10.1145/3093333.3009872
libuv. 2025. libuv: Cross-platform asynchronous I/O. https://libuv.org/ Accessed: 2025-02-28.
Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, NY, USA, 500–514. doi:10.1145/3009837.3009897
Matthew Lutze and Magnus Madsen. 2024. Associated Effects: Flexible Abstractions for Effectful Programming. *Proc. ACM Program. Lang.* 8, PLDI, Article 163 (June 2024), 23 pages. doi:10.1145/3656393
Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 330 (Oct. 2024), 29 pages. doi:10.1145/3689770
Moe Masuko and Kenichi Asai. 2009. Direct Implementation of Shift and Reset in the MinCaml Compiler. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML* (Edinburgh, Scotland) *(ML '09)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/1596627.1596636
Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 255 (oct 2023), 30 pages. doi:10.1145/3622831
Cao Nguyen Pham and Martin Odersky. 2024. Stack-Copying Delimited Continuations for Scala Native. In *Proceedings of the 19th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems* (Vienna, Austria) *(ICOOOLPS 2024)*. Association for Computing Machinery, New York, NY, USA, 2–13. doi:10.1145/3679005.3685979
Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable effects for flexible and accelerated probabilistic programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).
Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. 7, OOPSLA2, Article 238 (oct 2023), 26 pages. doi:10.1145/3622814
Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013
Ron Pressler. 2018. Multiprompt delimited continuations. https://mail.openjdk.org/pipermail/loom-dev/2018-September/000145.html. loom-dev mailing list.
Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 96–111. doi:10.1145/3453483.3454032

Amr Hany Saleh and Tom Schrijvers. 2016. Efficient algebraic effect handlers for Prolog. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 884–898. doi:10.1017/S147106841600034X

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2022. Region-based Resource Management and Lexical Exception Handlers in Continuation-Passing Style. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings* (Munich, Germany). Springer-Verlag, Berlin, Heidelberg, 492–519. doi:10.1007/978-3-030-99336-8_18

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

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

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, Article 71 (aug 2021), 30 pages. doi:10.1145/3473576

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (Nov. 2020), 30 pages. doi:10.1145/3428207

Received 2025-02-27; accepted 2025-06-27