Virtualizing Continuations

Effect handlers and multishot continuations are powerful abstractions for managing control flow; together, they offer concise and modular ways to express and handle nondeterminism, randomness, and more. However, implementing multishot continuations in the presence of stack-allocated lexical resources—lexical effect handlers in particular—is challenging, since stack copying invalidates references to these resources.

We present a novel implementation strategy for lexical effect handlers that fully supports multishot continuations. The key idea is to *virtualize* the stack space used by continuations. Each stack-allocated handler instance is assigned a virtual address, and all effect invocations through these virtual addresses are mediated by an address translation mechanism. A software-based memory management unit in the runtime system performs these translations efficiently, exploiting the lexical scoping discipline of effect handlers.

We capture the essence of our approach via a new operational semantics for lexical effect handlers and prove it correct with respect to the standard semantics. We also implement it in a compiler and runtime system. Compared to prior languages with lexical effect handlers, our implementation increases expressivity by fully supporting multishot continuations—and, as a happy consequence, unlocks significant performance gains by enabling parallel execution of multishot continuations.

1 Introduction

Effect handlers are a powerful language abstraction that enables structuring advanced control-flow patterns [15]. They separate the code that may raise an effect from the code that handles it, providing a modular way to manage computational effects such as exceptions, cooperative concurrency, and nondeterminism. Raising an effect suspends the current computation and captures the delimited continuation (a.k.a. *resumption*) up to the corresponding handler, in which the programmer can choose whether, when, and how many times to resume the continuation.

Lexical effect handlers. Like exception handlers, effect handlers in most languages are dynamically scoped: when an effect is raised, the dynamically closest enclosing handler with the same effect name is chosen to handle the effect. However, it has been shown that dynamically scoped handlers threaten abstraction safety and hinder modular reasoning [23], in the same way that dynamically scoped variables do. In response, some language-design efforts have shifted focus to *lexically scoped* handlers [23; 3; 21; 12; 7].

In these languages, a handler acts as a stack-allocated, lexically scoped resource (sometimes also referred to as a *capability*). Dynamically, the capability is created when control enters the lexical scope of the handler. Statically, only code that is passed the capability—either because it lies textually inside the handler's lexical scope or because it is passed the capability from its caller—can raise effects to the handler. The payoff is predictability and local reasoning principles: the programmer can reason modularly about where a raised effect propagates without fretting that some dynamically enclosing handler might accidentally intercept it.

Accordingly, recent research has investigated implementation techniques for lexical effect handlers. One particularly promising approach, based on stack switching, uses *stack addresses* as unique identifiers for handlers installed on the stack. This approach avoids searching the stack for the appropriate handler when an effect is raised: the runtime can directly locate the handler via its stack address, thereby enabling efficient handler lookup and continuation capture. The technique was first introduced in the Lexa language [12] and subsequently adopted by Effekt [13]. Unfortunately, this implementation strategy appears incompatible with multishot continuations.

The power of multishot resumptions. The ability to resume a continuation multiple times is an attractive language feature. It facilitates concise and modular implementations of parsing, game search, proof search, symbolic execution, probabilistic inference, checkpointing, and more.

```
1  effect Decision =
2  | choose (actions: list[Action]) : Action
3  | emit (reward: float) : unit

4  type Dist = list[(float, 'a)]
5  effect Probability =
6  | sample (dist: Dist['a]) : 'a

7  def step (d: Decision) (p: Probability) (s: State) : State =
8    let a = raise d.choose (actions s) in let s' = nextState p s a in
9    raise d.emit (reward s a s'); s'

10 def actions (s: State) : list[Action]
11 def nextState (p: Probability) (s: State) (a: Action) : State
12 def reward (s: State) (a: Action) (s': State) : float

13 def valueIter (n: int) (s: State) : float =
14   if n == 0 then 0.0
15   else
16     handle (d: Decision) (p: Probability) ⇒
17       valueIter (n - 1) (step d p s)
18     with
19     | choose choices k ⇒ List.max (List.map choices (fun c ⇒ resume k c))
20     | emit r k ⇒ r + 0.9 * resume k ()
21     | sample dist k ⇒ List.sum (List.map dist (fun (prob, s) ⇒ prob * resume k s))
```

Figure 1. Lines 1–12: A framework for defining MDPs, via the `Decision` and `Probability` effects. Lines 13–21: An algorithm for solving MDPs, via effect handlers for `Decision` and `Probability`. For brevity, `valueIter` does not check for convergence. The handlers use multishot resumptions (lines 19 and 21).

For example, Figure 1 shows how a simplified version of Bellman's value iteration algorithm [1] for sequential decision making can be implemented with the help of lexical effect handlers and multishot resumptions. Lines 1–12 give a generic framework for defining decision-making problems as Markov Decision Processes (MDPs). Lines 13–21 give an algorithm for solving MDPs.

The `step` function models a single step of an MDP: the decision-making agent chooses an action, which yields a next state and an immediate reward. The state transition function `nextState` is probabilistic; hence it needs the capability to perform the `Probability` effect. The `step` function needs capabilities for both `Decision` and `Probability`.

Bellman's value iteration algorithm is implemented by the `valueIter` function. It computes the *value* of a state `s` (i.e., the optimal expected cumulative reward) with a planning horizon of `n` steps, by recursively computing the value of the next state with a horizon of `n - 1` steps.

Raising an effect (e.g., lines 8 and 9) transfers control to the handler (e.g., lines 19 and 20), while also capturing the delimited continuation up to the handler. The handler capabilities that `step` uses to raise effects are bound to the variables `d` and `p` declared in line 16, and the handlers are defined in lines 19–21. They interpret the `Decision` and `Probability` effects, by exploring all possible choices of actions and all possible state transitions.

- The handler for `Decision` interprets `choose` by invoking the resumption `k` with every possible choice of action—multishot resumptions!—and taking the maximal reward.
- The handler for `Decision` also interprets `emit` by adding the immediate reward to the discounted future reward obtained by resuming the continuation once.
- The handler for `Probability` interprets `sample` by invoking the resumption `k` with every possible state from the distribution's support—again, multishot resumptions!—and computing the expected reward averaged over all possible state transitions.

Resuming a continuation transfers control back to the point where the effect was raised, restoring the state of the suspended computation. By allowing the continuation to be resumed multiple times,

```
19      | choose choices k ⇒ List.parMax (List.parMap choices (fun c ⇒ resume k c))
20      | emit r k ⇒ r + 0.9 * resume k ()
21      | sample dist k ⇒ List.parSum (List.parMap dist (fun (prob, s) ⇒ prob * resume k s))
```

Figure 2. A parallelized version of the handler code for the `choose` and `sample` operations. This code leads to run-time errors in both Lexa and Effekt.

multishot resumptions provide an elegant way to travel back in time and explore more than one possible future.

An implementation challenge. Multishot resumptions are powerful, but they pose a seemingly irreconcilable challenge for the stack-based implementation strategy found in Lexa and Effekt.

In such a stack-based implementation, multishot resumptions require making multiple copies of the captured continuation, each with its own copy of the same portion of the stack. This stack copying is known to be a problem when the stack contains pointers into itself: the copied stack frames contain pointers that still refer to locations in the *original* stack rather than the new copy.

This situation arises in Lexa and Effekt, as they use stack addresses to identify handlers installed on the stack. After copying, the handler references in the copied stack still point to handlers on the original stack.[1] Continuing to use these stale references causes incorrect behavior and potentially security vulnerabilities. It is possible to work around this problem by scanning the stack and fixing up all stale references, but this approach is costly and thus often avoided in practice.

As a result, Lexa outright disallows stacks captured for multishot resumptions to contain handlers; Lexa aborts the program if such a situation is detected at run time [12]. Effekt lifts this restriction by using a global pointer that routes handler references to the active resumption copy. However, this approach still disallows a captured continuation to be resumed while another copy of it is active; Effekt aborts the program if such a situation is detected at run time [13].

Importantly, this latter restriction, shared by both Lexa and Effekt, prevents parallel execution of multishot resumptions. We make an observation that is perhaps obvious but often overlooked in current research on effect handlers: many applications of multishot resumptions are *embarrassingly parallel*, meaning individual resumptions can be executed independently with little communication. For example, in Figure 1, the handling of both the nondeterministic choices of actions and the probabilistic outcomes of state transitions ought to be easily parallelizable: Figure 2 presents a parallelized version of the handler code where the different copies of the same resumption are run in parallel using `List.parMap`. This parallelization could significantly speed up the value iteration algorithm on modern multicore processors, when the state and action spaces are large. Unfortunately, the restrictions in Lexa and Effekt make them incompatible with such simple forms of parallelism, limiting the usefulness of multishot resumptions in practice.

Our contributions. We present a novel implementation strategy for lexical effect handlers that fully supports multishot resumptions without the aforementioned restrictions, by virtualizing resumptions.

We draw an analogy between managing multishot resumptions in a language runtime and managing multiple processes in an operating system (OS). An OS uses virtual memory to provide each process with a virtual view of its address space. Similarly, we hope to provide each copy of a resumption with a virtual view of the identities of the handlers installed within it, so that code

[1] This problem is not unique to lexical effect handlers. It also arises in other settings that involve stack copying and *stack-allocated* (i.e., lexical) resources, since stack allocation leads to pointers into the stack. We focus on lexical effect handlers in this paper, but the ideas presented here can be adapted to other settings as well. In fact, our implementation already supports some form of stack-allocated state using the same technique (Section 5.4).

running in the resumption copy need not care where stack copying has relocated the handlers, even when multiple copies of the same resumption are run in parallel.[2]

While modern OSes rely on hardware-based memory management units to perform address translation, our approach uses a software-based memory management unit to translate virtual handler references to concrete stack addresses that uniquely identify handlers on the stack.

A complicating difference from traditional OS virtual memory is that, while each OS process can access only its own address space, a resumption may access handlers on enclosing resumptions' stacks. Fortunately, the lexical scoping discipline of handlers enables static reasoning about which handlers are accessible to a resumption. This observation allows us to design an efficient address translation mechanism that walks a linked list of *zones*—rather than finer-grained stack frames or stack segments—thereby minimizing the run-time work needed for handler lookup.

The paper is organized as follows. Section 2 presents the main ideas from a language implementer's perspective. Section 3 formalizes the essential aspects of the new implementation strategy as a new operational semantics for lexical effect handlers, and Section 4 proves that the new semantics is correct with respect to the standard semantics. Section 5 gives a more detailed view of the implementation. The evaluation in Section 6 suggests that our new approach enables significant speedups for embarrassingly parallel multishot resumptions while incurring minimal overhead on zero- and single-shot resumptions. Section 7 discusses related work, and Section 8 concludes.

2 Main Ideas

2.1 Background: Handler Lookup in a Stack-Switching Runtime

A stack-switching implementation of effect handlers works by associating each installed handler with its own *stack segment* (a contiguous block of memory). We use the term *stack* to denote a linked list of stack segments; the linked list structure reflects dynamic nesting of handler installations. When an effect is raised, the current stack is unwound up to the handler responsible for handling that effect, and control jumps to that handler with the continuation packaged as a suspended stack composed of the unwound segments. If the handler later resumes the continuation, execution switches back to that suspended stack.

Therefore, the compiler must efficiently implement two operations: (1) finding the appropriate handler for a given effect, and (2) capturing, and later resuming, the continuation between the raise site and the handler. The second operation is easy to implement efficiently: once the correct handler is located, capturing the continuation just means recording the top of the suspended stack and switching the stack pointer. Resuming the continuation then restores the saved stack pointer, effectively reinstating the captured execution context.

We will thus focus on the first operation: handler lookup. With lexical effect handlers, a raise site is statically associated with a lexically scoped variable identifying a handler installed in the current stack. The job of handler lookup is to find this handler.

An example. To illustrate the semantics of lexical handlers, consider the program in Figure 3. For simplicity, we assume that the effect type `E` has only one operation `op : unit → unit`. Figure 4 shows the stack snapshots at four key program points.

When control enters a `handle` expression, a new handler instance is installed on the stack: the runtime allocates a new stack segment for the handler and assigns a globally unique name to the newly installed handler. These dynamically generated names are the run-time representation of the lexically scoped variables bound by `handle` expressions (`e0`, `e1`, `e2`, `e3`, and `e4` in Figure 3). The first

[2]We do not take the analogy too far. Our analogy pertains only to address space abstraction and the idea of address translation. The actual mechanisms for address translation differ. We do not attempt to claim analogies to other aspects of virtual memory, such as paging, either.

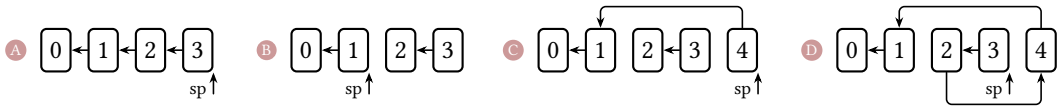


Figure 4. Stack snapshots at four program points during the execution of the program from Figure 3.

snapshot in Figure 4 shows the stack at program point A. Four handlers have been installed at this point, and they are assigned names 0, 1, 2, and 3. The runtime can choose these names arbitrarily as long as they are unique; we use small integers for illustration. The current stack consists of four segments. The arrow between two segments indicates the control flow when the computation on the deeper segment finishes and returns to the shallower one.

Next, upon `raise e2.op ()` on line 5, the runtime performs a handler lookup to find the handler that the variable `e2` refers to. Since `e2` has been substituted with the name 2, the runtime searches the stack for a handler with that name. Once found, the stack is unwound up to that handler, capturing a resumption stack consisting of the two unwound segments, and control jumps to program point B. The second diagram in Figure 4 captures the state of the stacks at this point.

As control enters the `handle` expression on line 9 (program point C), a new stack segment is allocated and a new handler instance (with name 4) is installed. The state of the stacks at this point is shown in the third diagram in Figure 4.

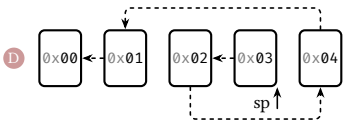
Next, on line 10, the captured continuation `k2` is resumed, so the suspended resumption stack is reinstated on top of the current stack. Control transfers to the program point immediately after the raise site (i.e., program point D). The rightmost diagram in Figure 4 shows the stack at this point.

```
1  handle (e0: E) ⇒
2    handle (e1: E) ⇒
3      handle (e2: E) ⇒
4        handle (e3: E) ⇒ A
5          raise e2.op (); D
6          raise e3.op ();
7        with op _ k3 ⇒ ...
8      with op _ k2 ⇒ B
9        handle (e4: E) ⇒ C
10         resume k2 ();
11         ...
12       with op _ k4 ⇒ ...
13   with op _ k1 ⇒ ...
14 with op _ k0 ⇒ ...
```

Figure 3. A program of 5 handlers.

Two implementations of handler lookup. One way to implement handler lookup is to traverse the stack, segment by segment, until a handler with a matching name is found. This approach exactly implements the semantics of lexical handlers, but incurs an overhead proportional to the stack depth. We call it the *search-based* implementation.

A more efficient implementation uses handlers' stack addresses as their identities [12; 13], depicted in the right diagram. Stack addresses like `0x02` are taken at the point of handler installation and passed down to raise sites. The dashed lines still indicate dynamic nesting of handler installations, but they are not needed for handler lookup. Instead, handler lookup is a constant-time operation: the runtime simply switches the stack pointer to the stack address passed down to the raise site. We call this implementation *address-based*.



Problem with identifying handlers by stack addresses. Unfortunately, this address-based implementation is simply incorrect in the presence of multishot resumptions.

Suppose that the resumption `k2` needs to be resumed twice (e.g., `k2` can be resumed again on line 11). The runtime must make a copy of the stack segments belonging to `k2`. In Figure 5, the snapshots in the top row are captured at program point D after a copy of the resumption stack is made and installed. In the search-based implementation, the handlers installed in the copied stack segments retain their original names (2 and 3). In contrast, in the address-based implementation, the handlers in the copied segments are associated with new addresses (`0x12` and `0x13`).

Immediately after program point D, the program executes `raise e3.op ()` on line 6. In Figure 5, the snapshots in the bottom row are captured immediately after control transfers to a handler.

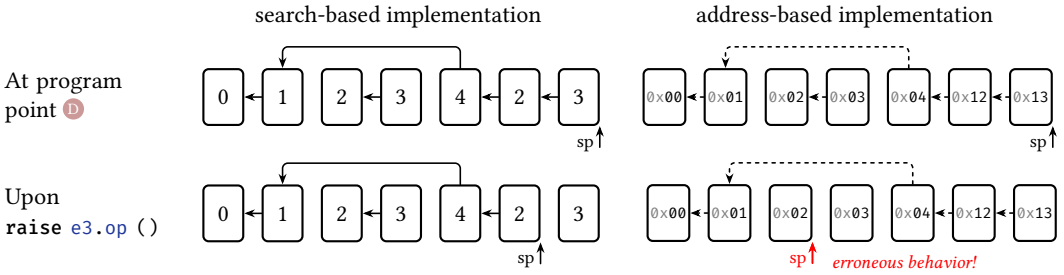


Figure 5. Stack snapshots before and after raising an effect within a copied resumption.

In the search-based implementation, the runtime correctly finds the handler with name 3 via stack traversal and switches to the stack segment immediately enclosing it. In the address-based implementation, however, program point D still holds the original address `0x03`, which is no longer valid—the stack segment associated with that address is not even part of the current stack. As a result, the runtime switches to the wrong segment.

A possible remedy is to scan the copied stack segments and update all references pointing into the original segments. This scan and repair is costly and thus often not considered a viable solution.

2.2 Virtualizing Stack Space

We present a new implementation strategy, which we call *virtual resumptions*, that preserves the efficiency of the address-based implementation while correctly handling multishot resumptions.

This new strategy, inspired by virtual memory management in OSes, can be viewed as a hybrid of the search- and address-based strategies. Like the address-based implementation, our new implementation still takes stack addresses at handler installation and passes them down to raise sites. But these addresses are now *virtual*.[3] When such a virtual address is used (e.g., at a raise site), the runtime performs address translation to obtain the actual stack address. A software-level memory management unit (sMMU) in the language runtime manages the virtualized stack space and performs address translation. This address translation works in a way akin to the search-based strategy, but more efficient.

Zones and virtual addresses. Stack segments live in *zones*, managed by the sMMU. The sMMU ensures that copies of the same stack segment live in distinct zones: each time a copied resumption is resumed, a new zone is allocated to hold the copied stack segments. Figure 6 shows that a new zone (Zone 1) is allocated when the resumption `k2` is copied and resumed.

One way to implement the sMMU is to allocate each zone as a large contiguous memory region and allocate stack segments within it as fixed-size blocks. We implement this approach in our compiler and runtime system (Section 5.1), but it is possible to implement zones and segments using other allocation strategies (Section 5.6). The formalisms in Section 3 give a more abstract account independent of these implementation details.

In our canonical implementation, a virtual address has three fields: the higher bits is called the *zone number*, middle bits the *handler ID*, and lower bits the *offset* (see Figure 7).

Upon entering a `handle` expression, the sMMU allocates a new stack segment in the current zone using the next available handler ID. This virtual address (with offset 0) marks the installation point of the handler; it is passed down the stack as the capability for invoking operations implemented by that handler. The next available handler ID is drawn from an incrementing global counter initialized to zero at program start. The offset field is used to index into the stack storage local to the handler.

[3]Terminology note: we will use the term *virtual* to exclusively refer to the stack-space virtualization provided by our runtime system. Beneath this layer of abstraction, the OS may also virtualize physical memory. Since OS virtual memory is transparent to our design conceptually, we do not consider it until Section 5 where we discuss implementation details.

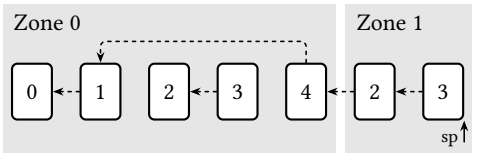


Figure 6. For clarity, stack segments are marked with handler IDs only. Full virtual addresses can be reconstructed by prepending the zone number and appending the offset.

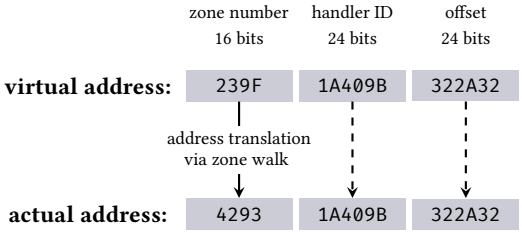


Figure 7. Structure of virtual addresses in the implementation described in Section 5. Address translation finds the actual zone number, via zone walk.

When a multishot resumption is resumed, the sMMU allocates a new zone, extending the linked list of zones already on the active stack. The stack segments belonging to the resumption are copied to the new zone. There is no need to scan the copied segments and fix up stale references. The same virtual addresses can be used, and they now refer to the handlers in the copied segments via address translation when used at raise sites.

Address translation via zone walk. The use of virtual addresses necessitates address translation, which locates the zone containing the target handler through a process called *zone walk*.

Upon `raise`, the sMMU receives a virtual address to translate. First, it extracts the handler ID H from the virtual address. It then determines the current zone number Z using the stack pointer sp. After that, it walks the linked list of zones backwards, starting from the current zone Z.

Zone walk is performed at the granularity of zones. As the runtime traverses each zone, it does *not* examine every handler in the zone and compare its ID with H, which would be too expensive. Instead, it uses a simple *presence check*: for a given zone, if the first handler in the zone has ID H' such that $H \geq H'$, it must be the case that the zone contains handler H, so the walk stops. Otherwise, the walk continues to the previous zone. It may be surprising that this simple check suffices. Its correctness, however, rests on two key properties of the language: (1) the runtime generates monotonically increasing handler IDs, and (2) handlers in the program text are lexically scoped. They allow us to establish the correctness of zone walk in Section 4.

Once the zone walk stops, say, in a zone Z', the sMMU completes the address translation: it modifies the original virtual address by replacing the zone number Z with Z' (see Figure 7). This actual address is then returned to the program for use in switching the stack pointer.

Demonstration. The stack snapshot in Figure 6 is taken at program point D right after the resumption `k2` is copied and resumed. Upon `raise e3.op ()` on line 6, the sMMU receives a virtual address—zone number $Z = 0$, handler ID $H = 3$—to translate. From the current sp, the sMMU determines that the current zone number is 1. The sMMU then begins the zone walk. It first checks whether Zone 1 contains a handler with ID 3 using the presence check described earlier: since the first handler in Zone 1 has ID 2, and $H \geq 2$, the sMMU concludes that Zone 1 contains the desired handler. The sMMU immediately constructs the actual address by replacing the zone number Z in the virtual address with 1 and returns the actual address to the program.

As another example, suppose line 6 in Figure 3 is `raise e1.op ()` instead. The sMMU now needs to translate a virtual address with zone number $Z = 0$ and handler ID $H = 1$. Because the first handler in Zone 1 has ID 2, and $H < 2$, the zone walk continues to Zone 0. Now, the first handler in Zone 0 has ID 0. Because $H \geq 0$, the sMMU determines it has found the correct zone, so it replaces the zone number Z in the virtual address with 0, which happens to be the same as Z, and returns this translated address.

Discussion: correctness and efficiency. An attentive reader may notice that the zone walk fails when translating a virtual address with zone number 0 and handler ID 4: because the first handler in Zone 1 has ID 2, and $4 \geq 2$, the walk would stop in Zone 1—but Zone 1 has no handler with ID 4! Rest assured, this scenario is not actually possible. While Figure 6 might give the impression that code running in Zone 1 could access the handler with ID 4, the type system prevents this possibility, thanks to the lexical scoping of handlers. Zone 1 is created for a copy of the resumption `k2`, which is captured between program points Ⓐ and Ⓑ. At that point, variable `e4` is not lexically in scope, so the handler with ID 4, which is only installed later, is inaccessible from within the resumption `k2`.

Zone walk is efficient. If the program does not use multishot resumptions, our implementation creates no new zones (Section 5.2), so all stack segments live in a single zone—zone walk reduces to a constant-time address-based lookup. When multishot resumptions are used, the overhead of zone walk is in proportion to the number of zones traversed. Even so, zone walk remains more efficient than the search-based alternative, since traversal proceeds zone by zone rather than segment by segment, and the number of zones traversed is typically a small constant in practice.

Virtual resumptions, unlike existing implementations of lexical effect handlers [12; 13], allow multiple copies of the same resumption to be actively running at the same time, as address translation through zone walk does not depend on centralized resources. This ability has a happy consequence: the many copies of the same resumption can run in parallel without interfering with each other, which may translate to significant performance gains on multicore machines.

At this point, the reader may not believe that zone walk is guaranteed to find the correct handler, especially because the presence check used by the walk may appear too coarse. In Section 4, we prove that virtual resumptions indeed implement the semantics of lexical effect handlers. But first, let us formalize the ideas in Section 3.

3 A Formal Model of the Zone-Based Semantics

In this section, we capture the essence of virtual resumptions formally. We present two operational semantics for a core language with lexical effect handlers: a standard semantics (SL) where raising an effect searches the stack for the nearest handler with a matching name, and a zone-based semantics (ZL) where the handler is found by walking the stack of zones. Programs written for SL can be executed under ZL without modification.

The two semantics differ only in their treatment of continuation delimiters. ZL introduces a new form of delimiter—zone delimiters—that marks zone boundaries. Section 4 proves a simulation theorem between SL and ZL. Section 5 covers implementation details not modeled by ZL.

3.1 Syntax

The syntax of our core language is given below. It is a call-by-value lambda calculus extended with lexical effect handlers.

$$\begin{array}{llll} \text{value} & v ::= x \mid () \mid \lambda x.\, e & \text{expression} \quad e ::= & v \mid e_1\ e_2 \mid \texttt{handle}\ \lambda x.\, e\ \texttt{with}\ h \mid \\ \text{handler value} & h ::= \lambda(y, k).\, e & & \texttt{raise}\ e_1\ (e_2) \mid \texttt{resume}\ e_1\ (e_2) \end{array}$$

In the expression $\texttt{handle}\ \lambda x.\, e\ \texttt{with}\ h$, the lambda abstraction $\lambda x.\, e$ is the effectful computation under the handler h. It binds a lexically scoped variable x standing for the handler name (aka *label*) freshly generated at run time when control enters e. The handler h, which is another lambda, binds two variables: y for the payload of the raised effect and k for the resumption. In the expression $\texttt{raise}\ e_1\ (e_2)$, e_1 identifies the handler to which the effect is raised, and e_2 is the effect payload. In the expression $\texttt{resume}\ e_1\ (e_2)$, e_1 stands for the resumption, and e_2 is the input to the resumption.

We make the standard simplifications that each handler handles exactly one effect, that each effect signature contains exactly one operation, and that each operation has exactly one argument.

$$
\begin{array}{rrcl}
\text{value} & v & ::= & \cdots \mid \texttt{lbl}(n) \mid \texttt{cont}\ K \\
\text{delimiter} & \omega & ::= & \bowtie \mid \blacktriangleright\!\blacktriangleleft \\
\text{frame} & A & ::= & \square\, e \mid v\, \square \mid \texttt{raise}\ \square\,(e) \mid \texttt{raise}\ v(\square) \mid \texttt{resume}\ \square\,(e) \mid \texttt{resume}\ v(\square) \\
\text{stack} & K & ::= & \epsilon \mid K :: A \mid K :: \omega_h^n \\
\text{zone} & Z & ::= & \epsilon \mid Z :: A \mid Z :: \bowtie\!\!\!\!\!\!\blacktriangleleft_h^n \\
\text{configuation} & M & ::= & \langle K \parallel e \parallel m \rangle \parallel \langle K \parallel K' \parallel n \parallel v \parallel m \rangle
\end{array}
$$

$$\mathcal{L}(\epsilon) = \emptyset \qquad \mathcal{L}(K :: A) = \mathcal{L}(K) \qquad \mathcal{L}(K :: \omega_h^n) = \mathcal{L}(K) \cup \{n\}$$

Figure 8. Auxiliary syntax for the operational semantics. Elements in orange are specific to ZL. The rest are shared by both semantics.

$$
\begin{array}{rll}
\textsc{app} & \langle K \parallel e_1\ e_2 \rangle & \longrightarrow \langle K :: \square\ e_2 \parallel e_1 \rangle \\
\textsc{beta} & \langle K \parallel (\lambda x.\, e)\ v \rangle & \longrightarrow \langle K \parallel e\{v/x\} \rangle \\
\textsc{enter} & \langle K \parallel \texttt{handle}\ \lambda x.\, e\ \texttt{with}\ h \parallel m \rangle & \longrightarrow \langle K :: \bowtie\!\!\!\!\!\!\blacktriangleleft_h^m \parallel e\{\texttt{lbl}(m)/x\} \parallel m+1 \rangle \\
\textsc{leave1} & \langle K :: \omega_h^n \parallel v \rangle & \longrightarrow \langle K \parallel v \rangle \\
\textsc{leave2} & \langle K :: A \parallel v \rangle & \longrightarrow \langle K \parallel A\,[v] \rangle \\
\textsc{raise} & \langle K_1 :: \bowtie\!\!\!\!\!\!\blacktriangleleft_h^n :: K_2 \parallel \texttt{raise}\ \texttt{lbl}(n)\ (v) \rangle & \longrightarrow \langle K_1 \parallel \bowtie\!\!\!\!\!\!\blacktriangleleft_h^n :: K_2 \parallel n \parallel v \rangle \\
 & \text{where } n \notin \mathcal{L}(K_2) & \\
\textsc{raise} & \langle K \parallel \texttt{raise}\ \texttt{lbl}(n)\ (v) \rangle & \longrightarrow \langle K \parallel \epsilon \parallel n \parallel v \rangle \\
\textsc{unw-inter-zone} & \langle K :: \blacktriangleright\!\blacktriangleleft_h^m :: Z \parallel K' \parallel n \parallel v \rangle & \longrightarrow \langle K \parallel \blacktriangleright\!\blacktriangleleft_h^m :: Z :: K' \parallel n \parallel v \rangle \\
 & \text{where } m \geq n & \\
\textsc{unw-intra-zone} & \langle K :: \bowtie\!\!\!\!\!\!\blacktriangleleft_h^n :: Z \parallel K' \parallel n \parallel v \rangle & \longrightarrow \langle K \parallel \blacktriangleright\!\blacktriangleleft_h^n :: Z :: K' \parallel n \parallel v \rangle \\
\textsc{invoke} & \langle K \parallel \omega_h^n :: K' \parallel n \parallel v \rangle & \longrightarrow \langle K \parallel e\{v/x\}\{\texttt{cont}\ \omega_h^n :: K'/k\} \rangle \\
 & \text{where } h = \lambda(x,k).\, e & \\
\textsc{resume} & \langle K \parallel \texttt{resume}\ (\texttt{cont}\ K')\ (v) \rangle & \longrightarrow \langle K :: K' \parallel v \rangle
\end{array}
$$

Figure 9. Reduction rules of SL and ZL. Rules in blue are specific to SL, those in orange are specific to ZL, and the rest are shared by both semantics. Counter m is omitted when not relevant to the reduction step.

3.2 Operational Semantics

Figure 8 shows the auxiliary syntax. Figure 9 presents two small-step abstract-machine semantics for the core language. The syntax and reduction rules shared by both semantics are shown in black, those specific to ZL are shown in orange, and those specific to SL are shown in blue.

Auxiliary Syntax. The syntax of values is extended with labels $\texttt{lbl}(n)$, which serve as handler identities at run time, and continuations $\texttt{cont}\ K$. A *frame* A is a computation with a hole $\square$ in it. Frames can be concatenated to form a *stack* K, with the next frame plugged into the hole of the previous frame. There is a special kind of frame ω_h^n, called *handler frame*, that represents a handler with label n and implementation h installed on the stack. The delimiter ω is either a standard delimiter $\bowtie$ or a zone delimiter $\blacktriangleright\!\blacktriangleleft$; zone delimiters exist only in ZL. The empty stack is denoted by ϵ.

A configuration M is either in *normal mode* or *unwind mode*. In normal mode, it consists of a stack K, an expression e, and a counter m used for generating fresh labels. The counter m is incremented whenever a new handler is installed. A configuration in unwind mode has the form $\langle K \parallel K' \parallel n \parallel v \parallel m \rangle$, where K is the part of the stack yet to be unwound, K' is the partial resumption stack constructed so far, n is the label of the target handler, and v is the effect payload. We will omit the counter m in configurations when it is not relevant to the reduction step.

A *zone* Z is a special case of a stack K. A stack may contain handler frames with both kinds of delimiters $\bowtie$ and $\blacktriangleright\!\blacktriangleleft$, while a zone may *not* contain handler delimiters $\blacktriangleright\!\blacktriangleleft$. A handler delimiter $\blacktriangleright\!\blacktriangleleft$ marks the beginning of a new zone, so we will abuse terminology and refer to both Z and $\blacktriangleright\!\blacktriangleleft_h^n :: Z$ as zones. The auxiliary function $\mathcal{L}$ computes the set of labels present in a stack.

Reduction rules. Figure 9 presents the reduction rules for both semantics. All administrative rules are omitted for brevity, except for APP, which is shown as an example.

Rule ENTER installs a new handler frame on the stack. The handler is given a fresh label m, and the counter is incremented immediately afterwards. The label $\texttt{lbl}(m)$ then substitutes for the variable x, and control enters e. Notice that in both SL and ZL, a handler frame, when initially installed, is always delimited by $\bowtie$. While the delimiter remains $\bowtie$ in SL, in ZL, it may be upgraded to $\blacktriangleright\!\blacktriangleleft$ later when a continuation delimited by it is captured.

The semantics SL and ZL differ in how they reduce $\texttt{raise lbl}(n)\ (v)$. SL's RAISE rule splits the stack into two parts at the nearest handler frame with label n. The resulting configuration, now in unwind mode, is *invoke-ready*, as the desired handler has been found and the resumption fully constructed. Rule INVOKE now takes over. It invokes the handler, substituting the constructed resumption for k. The configuration returns to normal mode.

ZL's RAISE rule does not immediately yield an invoke-ready configuration. Instead, to model zone walk (Section 2), it simply transitions the machine state into unwind mode, initiating an unwinding process that traverses the stack of zones to find the desired handler.

UNW-INTER-ZONE advances the traversal one zone at a time. If the current zone is guarded by a zone delimiter whose label m is greater than or equal to the desired label n, the zone $\blacktriangleright\!\blacktriangleleft_h^m :: Z$ is popped off the active stack and pushed onto the resumption stack. In particular, if $m = n$, the resulting configuration is invoke-ready, so zone walk ends here. Otherwise, zone walk continues.

UNW-INTRA-ZONE is in effect when the current zone contains a handler with label n. Crucially, the delimiter $\bowtie$ of the handler is upgraded to $\blacktriangleright\!\blacktriangleleft$, marking the creation of a new zone for the captured resumption. The resulting configuration is now invoke-ready, so zone walk concludes here as well.

Discussion. Although SL finds the desired handler in just one reduction step, a proper implementation of this semantics must still examine, one by one, the labels attached to each handler delimiter ($\bowtie$ or $\blacktriangleright\!\blacktriangleleft$) on the stack. In contrast, although handler lookup in ZL may take multiple unwinding steps, the number of steps is proportional only to the number of zone delimiters ($\blacktriangleright\!\blacktriangleleft$), which are typically far sparser on the stack than standard handler delimiters ($\bowtie$).

Also notice that ZL does not model zone numbers. Their exact values are immaterial to the high-level semantics of virtual resumptions. Instead, ZL models zone boundaries using zone delimiters; address translation is modeled as searching the stack for the right zone boundary via zone walk.

Demonstration. Consider the stack diagram in Figure 6 from Section 2. The shape of the same *active* stack in ZL is $\epsilon :: \bowtie^0 :: \bowtie^1 :: \bowtie^4 :: \blacktriangleright\!\blacktriangleleft^2 :: \bowtie^3$. Non-handler frames are omitted for brevity.

We demonstrate the semantics of ZL by two examples. In the first example, the effect is raised to a handler with ID 3. The reduction steps taken in ZL are RAISE, UNW-INTRA-ZONE, and then INVOKE—the desired handler is immediately found in the current zone. In the second example, the effect is raised to a handler with ID 1. The reduction steps taken in ZL are RAISE, UNW-INTER-ZONE, UNW-INTRA-ZONE, and INVOKE—first walking out of the current zone and then finding the desired handler in the enclosing zone.

4 Correctness

We now prove that ZL preserves the semantics of SL. This result justifies that the same program written with SL in mind behaves the same when run under ZL. This property is formally stated in Theorem 1, which says that if a program e type-checks and evaluates to a value v under SL, then it will also evaluate to the same value v under ZL. Recall that SL and ZL share the same source syntax.

THEOREM 1. *If* $\epsilon \mid \epsilon \mid \epsilon \vdash e : \tau$ *and* $\langle \epsilon \parallel e \parallel 0 \rangle \longrightarrow^* \langle \epsilon \parallel v \parallel m \rangle$, *then* $\langle \epsilon \parallel e \parallel 0 \rangle \longrightarrow^* \langle \epsilon \parallel v \parallel m \rangle$.

Theorem 1 follows easily from a simulation lemma (Lemma 1), which states that every reduction step in SL can be simulated by a sequence of steps in ZL.

Lemma 1 (Simulation). *Given a SL configuration M and a ZL configuration M such that $\vdash M$ and $M \sim \mathit{M}$, if $M \longrightarrow M'$, then there exists a ZL configuration M' such that $\mathit{M} \longrightarrow^* \mathit{M}'$ and $M' \sim \mathit{M}'$.*

The lemma is proved by case analysis on the SL reduction $M \longrightarrow M'$. Since most reduction rules are shared by SL and ZL, the argument is immediate in almost all cases. We therefore focus on the only nontrivial case, RAISE, whose proof is given at the end of this section. Before that, we first define the simulation relation $\sim$ (Section 4.1) and the type system (Section 4.2).

4.1 Relating SL and ZL

Lemma 1 is stated in terms of a relation $M \sim \mathit{M}$ between SL and ZL configurations. This relation is defined inductively over the syntax, which further requires relations over values, expressions, handler values, frames, and stacks: $v \overset{\text{val}}{\sim} \mathit{v}$, $e \overset{\text{exp}}{\sim} \mathit{e}$, $h \overset{\text{hdl}}{\sim} \mathit{h}$, $A \overset{\text{frm}}{\sim} \mathit{A}$, and $K \overset{\text{ctx}}{\sim} \mathit{K}$. Since the syntaxes of SL and ZL largely coincide, most of the rules are trivial and omitted. The only nontrivial case is the relation on stacks, two rules of which are shown to the right. The rules state that both forms of ZL delimiters $\bowtie_h^n$ and $\blacktriangleright\!\blacktriangleleft_h^n$ are related to the SL handler delimiter $\bowtie_h^n$.

$$\frac{K \overset{\text{ctx}}{\sim} \mathit{K} \qquad h \overset{\text{hdl}}{\sim} \mathit{h}}{K :: \bowtie_h^n \overset{\text{ctx}}{\sim} \mathit{K} :: \bowtie_h^n}$$

$$\frac{K \overset{\text{ctx}}{\sim} \mathit{K} \qquad h \overset{\text{hdl}}{\sim} \mathit{h}}{K :: \bowtie_h^n \overset{\text{ctx}}{\sim} \mathit{K} :: \blacktriangleright\!\blacktriangleleft_h^n}$$

4.2 Type System

Theorem 1 is stated for well-typed programs. Our type system is similar to prior type systems for lexical handlers [23; 25; 3; 4]. It applies to both source programs and run-time configurations.

Type-level constructs. The syntax of type-level constructs are given in Figure 10. In addition to types, *labels* and *capabilities* are also at the type level. We call them *constructors* collectively. A constructor context Δ tracks the kinds of *constructor variables*.

A label η is either a label variable ρ or a label constant n. Label constants exist only at run time.

A capability set C is composed of capability variables α and mappings from labels η to effect names F. For a program point that provides capabilities C, the presence of a mapping $\eta \mapsto F$ in C indicates that a handler for the F effect with label η is accessible from that program point.

Types include the `unit` type, label types, function types, and continuation types. A function type $\forall\,[\Delta \mid C]\,.\,\tau_1 \to \tau_2$ abstracts over constructor variables (bound in Δ). It also mentions a precondition C that describes the capabilities required to be present on the stack when the function is called. Dually, this C can also be seen as the side effects that the function may perform when called. Similarly, for a continuation of the type $\texttt{cont}\,[C]\,.\,\tau_1 \to \tau_2$, capabilities C must be present when it is invoked.

Selected typing rules. Figure 10 presents selected typing rules. For brevity, we omit obvious rules such as well-formedness rules for type-level constructs and subsumption rules. The full set of typing rules can be found in an appendix. The typing rules are implicitly parameterized over a global map $\mathbb{F}$ from effect names to their signatures.

Expression typing judgments have the form $\Delta \mid \Gamma \vdash e : \tau \mid C$, where C is the capabilities required to be present when evaluating e. The capabilities C can also be seen as the side effects that e may perform. Value typing judgments have the form $\Delta \mid \Gamma \vdash v : \tau$. Values are free of side effects.

Lambda abstractions are implicitly polymorphic in constructor variables (bound by Δ' in the typing rule). Label values $\texttt{lbl}(n)$ are given singleton types $\texttt{LBL}(n)$. Continuations are typed based on the type of the captured stack K.

In the typing rule for applications, constructor variables $\overline{\beta}$ parameterizing the function are instantiated by constructors $\overline{c}$. Capabilities C are required to be present when evaluating both e_1 and e_2 and when calling the function.

$$
\begin{array}{llll}
\text{constructor kind} & \kappa & ::= & \texttt{Label} \mid \texttt{Cap} \\
\text{constructor} & c & ::= & \eta \mid C \\
\text{label} & \eta & ::= & \rho \mid n \\
\text{capability set} & C & ::= & \epsilon \mid C \cup \alpha \mid \\
 & & & C \cup \{\eta \mapsto F\}
\end{array}
\qquad
\begin{array}{llll}
\text{constructor var} & \beta & ::= & \rho \mid \alpha \\
\text{constructor context} & \Delta & ::= & \epsilon \mid \Delta, \beta : \kappa \\
\text{term var context} & \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \\
\text{type} & \tau & ::= & \texttt{unit} \mid \texttt{LBL}(\eta) \mid \forall\,[\Delta \mid C]\,.\,\tau \to \tau \mid \\
 & & & \texttt{cont}\,[C]\,.\,\tau \to \tau
\end{array}
$$

label variable ρ label constant n
capability variable α effect name F

$$\texttt{dom}(\epsilon) = \emptyset \qquad \texttt{dom}(C \cup \{n \mapsto F\}) = \texttt{dom}(C) \cup \{n\}$$

$$\boxed{\Delta \mid \Gamma \vdash v : \tau}$$

$$\frac{\Delta, \Delta' \mid \Gamma, x : \tau_1 \vdash e : \tau_2 \mid C}{\Delta \mid \Gamma \vdash \lambda x.e : \forall\,[\Delta' \mid C]\,.\,\tau_1 \to \tau_2} \qquad \frac{}{\Delta \mid \Gamma \vdash \texttt{lbl}\,(n) : \texttt{LBL}(n)} \qquad \frac{\vdash K : [C']\,\tau_1 \rightsquigarrow [C]\,\tau_2}{\Delta \mid \Gamma \vdash \texttt{cont}\ K : \texttt{cont}\,[C]\,.\,\tau_1 \to \tau_2}$$

$$\boxed{\Delta \mid \Gamma \vdash e : \tau \mid C}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \left(\forall\left[\overline{\beta : \kappa} \middle| C\right].\,\tau_1 \to \tau_2\right) \mid C \qquad \Delta \mid \Gamma \vdash e_2 : \tau_1\left\{\overline{c}/\overline{\beta}\right\} \mid C \qquad \Delta \vdash \overline{c} : \overline{\kappa}}{\Delta \mid \Gamma \vdash e_1\ e_2 : \tau_2\left\{\overline{c}/\overline{\beta}\right\} \mid C} \qquad \frac{\Delta, \rho : \texttt{Label} \mid \Gamma, x : \texttt{LBL}\,(\rho) \vdash e : \tau \mid C \cup \{\rho \mapsto F\} \qquad \Delta \mid \Gamma \vdash h : F\,\#\,\tau \mid C \qquad \Delta \vdash \tau}{\Delta \mid \Gamma \vdash \texttt{handle}\ \lambda x.e\ \texttt{with}\ h : \tau \mid C}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \texttt{LBL}(\eta) \mid C \qquad C(\eta) = F \qquad \Delta \mid \Gamma \vdash e_2 : \tau_1 \mid C \qquad \mathbb{F}\,(F) = \tau_1 \to \tau_2}{\Delta \mid \Gamma \vdash \texttt{raise}\ e_1\,(e_2) : \tau_2 \mid C} \qquad \frac{\Delta \mid \Gamma \vdash e_1 : (\texttt{cont}\,[C]\,.\,\tau_2 \to \tau_{\text{ans}}) \mid C \qquad \Delta \mid \Gamma \vdash e_2 : \tau_2 \mid C}{\Delta \mid \Gamma \vdash \texttt{resume}\ e_1\,(e_2) : \tau_{\text{ans}} \mid C}$$

$$\boxed{\Delta \mid \Gamma \vdash h : F\,\#\,\tau \mid C}$$

$$\frac{\Delta \mid \Gamma, x : \tau_1, k : \texttt{cont}\,[C]\,.\,\tau_2 \to \tau_{\text{ans}} \vdash e : \tau_{\text{ans}} \mid C \qquad \mathbb{F}\,(F) = \tau_1 \to \tau_2}{\Delta \mid \Gamma \vdash \lambda(x, k).\,e : F\,\#\,\tau_{\text{ans}} \mid C}$$

$$\boxed{\vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C} \qquad \boxed{\vdash K : [C_1]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2}$$

$$\frac{\vdash K : [C_2]\,\tau_2 \rightsquigarrow [C_3]\,\tau_3 \qquad \vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C_2}{\vdash K :: A : [C_2]\,\tau_1 \rightsquigarrow [C_3]\,\tau_3} \qquad \frac{\vdash K : [C_1]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2 \qquad \Delta \mid \Gamma \vdash h : F\,\#\,\tau_1 \mid C_1 \qquad \forall m \in \texttt{dom}\,(C_1), m < n}{\vdash K :: \omega_h^n : [C_1 \cup \{n \mapsto F\}]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2}$$

Figure 10. Type-level constructs and selected typing rules. The type system works for both SL and ZL.

In the typing rule for `handle` $\lambda x.e$ `with` h, the expression e is parameterized over a label variable ρ and a term variable x of type $\texttt{LBL}(\rho)$. Variable x stands for the label value that will be dynamically created when the handler is installed. Variable ρ appears in the mapping $\rho \mapsto F$ in the capability context of e, which states the requirement that a handler for F be available when evaluating e. The premise $\Delta \mid \Gamma \vdash h : F\,\#\,\tau \mid C$ says that the handler value h implements the F effect and, when invoked, performs a computation having type τ and requiring capabilities C. The premise $\Delta \vdash \tau$ says that the type τ of the expression is well-formed without ρ in the constructor context, effectively preventing ρ and x from escaping. This condition is standard; type systems for lexical handlers [23; 3] or lexical regions [19; 6; 9] use it to ensure type safety.

In the typing rule for `raise` $e_1(e_2)$, e_1 is given the singleton type $\texttt{LBL}(\eta)$, and the capability context C must provide the capability to use η. The use of singleton types for dynamically created resources, as well as the separation between labels and the capabilities to use them, is similar to alias types [20].

In the typing rule for `resume` $e_1(e_2)$, the capabilities required to invoke the continuation e_1 must be present in the current capability context C.

Stack typing judgments have the form $\vdash K : [C_1]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2$. The type indicates that K can be installed on a stack which expects in its hole a computation of type τ_2 with capability context C_2. It also indicates that K itself expects in its hole a computation of type τ_1 with capability context C_1.

To type a stack of the form $K :: A$, the rule ensures that the output type and capability context of the frame A matches the input type and capability context of the stack K. Notice that the frame type of A does not have an input capability context, as non-handler frames do not introduce capabilities.

To type a stack of the form $K :: \omega_h^n$, it is additionally required that any capability needed by the computation to be installed in the hole of $K :: \omega_h^n$ be no arithmetically larger than n—that is, created no later than label n. This invariant holds because, statically, the computation can only access labels lexically in scope, and, dynamically, labels in the computation's lexical context are assigned monotonically increasing numbers when created.

Configuration typing $\vdash M$ is standard and thus omitted.

Type safety of SL. With the type system in place, we now state the type-safety theorem for SL. The type safety of similar systems is well-established [23; 3; 4]; we omit the proof here.

Theorem 2 (Type Safety). (Progress) *If* $\vdash M$*, then either* M *is a final configuration or there exists* M' *such that* $M \longrightarrow M'$. (Preservation) *If* $\vdash M$ *and* $M \longrightarrow M'$*, then* $\vdash M'$.

4.3 Proving Simulation

The simulation proof relies on a notion of labels *accessible* from the top of a stack:

$$\boxed{\mathtt{Acc}_m(K) = \left\{ l < m \;\middle|\; \text{Label } l \text{ is installed on stack } K \text{ and accessible from the top of } K \right\}}$$

$$\begin{aligned}
\mathtt{Acc}_m(\epsilon) &= \emptyset \\
\mathtt{Acc}_m(Z) &= \{l \in \mathcal{L}(Z) \mid l < m\} \\
\mathtt{Acc}_m(K :: \blacktriangleright\!\!\blacktriangleleft_h^n :: Z) &= \{l \in \mathcal{L}(\blacktriangleright\!\!\blacktriangleleft_h^n :: Z) \mid l < m\} \cup \mathtt{Acc}_{\min(n,m)}(K)
\end{aligned}$$

The definition is by a case analysis on the shape of the stack K: (1) K is empty, (2) K installs no zone delimiter, or (3) K installs at least one zone delimiter, and the nearest one to the top is $\blacktriangleright\!\!\blacktriangleleft_h^n$.

As a demonstration, consider again the active stack in Figure 6. The shape of the stack in ZL is $\epsilon :: \bowtie^0 :: \bowtie^1 :: \bowtie^4 :: \blacktriangleright\!\!\blacktriangleleft^2 :: \bowtie^3$. The set of labels accessible from the top of this stack is $\{0, 1, 2, 3\}$:

$$\begin{aligned}
\mathtt{Acc}_\infty(\epsilon :: \bowtie^0 :: \bowtie^1 :: \bowtie^4 :: \blacktriangleright\!\!\blacktriangleleft^2 :: \bowtie^3) &= \{a \in \{2,3\} \mid a < \infty\} \cup \mathtt{Acc}_{\min(\infty,2)}(\epsilon :: \bowtie^0 :: \bowtie^1 :: \bowtie^4) \\
&= \{a \in \{2,3\} \mid a < \infty\} \cup \{a \in \{0,1,4\} \mid a < \min(\infty, 2)\} = \{2,3,0,1\}
\end{aligned}$$

Notice that $\bowtie^4$ is not accessible from the stack top, because of the zone delimiter $\blacktriangleright\!\!\blacktriangleleft^2$ in between. If somehow $\bowtie^4$ could be invoked during evaluation, then no unwinding rule in Figure 9 would apply, and thus evaluation would get stuck.

We state some useful lemmas about accessible labels. Lemma 2 can be proved by induction on K.

Lemma 2. *If* $l \in \mathtt{Acc}_m(K)$*, then* $l < m$.

Lemma 3 states that, for a well-typed stack K, all the labels K makes available to its hole are accessible from within the hole. Lemma 3 can be proved by induction on the typing derivation of K.

Lemma 3. *If* $\vdash K : [C_1]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2$*, then* $\mathtt{dom}(C_1) \subseteq \mathtt{Acc}_\infty(K)$.

Lemma 4 states that, in a well-typed configuration that is about to invoke a handler with label n, the label is accessible from the top of the current stack. This is proved by applying inversion on the typing judgment, and then applying the two auxiliary lemmas.

Lemma 4. *If* $\vdash \langle K \parallel \mathtt{raise}\ \mathtt{lbl}(n)\ (v) \rangle$ *, then* $n \in \mathtt{Acc}_\infty(K)$.

Proof of Lemma 4. Because the configuration is well-typed, we have $\vdash K : [C_1]\,\tau_1 \rightsquigarrow [C_2]\,\tau_2$ and $\epsilon \mid \epsilon \vdash \mathtt{raise}\ \mathtt{lbl}(n)\ (v) : \tau_1 \mid C_1$. By the well-typedness of K and by Lemma 3, we have

$\mathtt{dom}(C_1) \subseteq \mathtt{Acc}_\infty(K)$. By the well-typedness of the `raise` expression, we have $n \in \mathtt{dom}(C_1)$. The result $n \in \mathtt{Acc}_\infty(K)$ follows. □

We are now ready to prove the RAISE case of Lemma 1, as promised earlier.

Proof of Lemma 1 (case RAISE). The proof in this case is about the following simulation diagram.

$$\begin{array}{ccc} \langle K_1 :: \bowtie_h^n :: K_2 \parallel \mathtt{raise}\ \mathtt{lbl}(n)\ (v)\rangle & \longrightarrow_{\mathrm{SL}} & \langle K_1 \parallel \bowtie_h^n :: K_2 \parallel n \parallel v\rangle \\ \sim & & \sim \\ \langle K_1 :: \omega_h^n :: K_2 \parallel \mathtt{raise}\ \mathtt{lbl}(n)\ (v)\rangle & \longrightarrow^*_{\mathrm{ZL}} & \langle K_1 \parallel \blacktriangleright\!\blacktriangleleft_h^n :: K_2 \parallel n \parallel v\rangle \end{array}$$

The top row is the SL reduction step by rule RAISE. The bottom row is the desired ZL reduction steps simulating the SL step. The goal is to show that given the SL configurations on the top-left and top-right, and the ZL configuration on the bottom-left, if they satisfy the relations $\longrightarrow_{\mathrm{SL}}$ (top row) and $\sim$ (left column), then there exists a ZL configuration on the bottom-right that satisfies $\longrightarrow^*_{\mathrm{ZL}}$ (bottom row) and $\sim$ (right column).

We show that a bottom-right ZL configuration exists. Its form is given in the diagram. This ZL configuration obviously satisfies the $\sim$ relation (right column). Now we derive $\longrightarrow^*_{\mathrm{ZL}}$ (bottom row). By rule RAISE, the bottom-left configuration first transitions into a configuration (name it M_1) in unwind mode: $M_1 = \langle K_1 :: \omega_h^n :: K_2 \parallel \epsilon \parallel n \parallel v\rangle$. Now, there are two cases to consider.

- K_2 contains no zone delimiters. So M_1 takes a single step to the bottom-right configuration by rule UNW-INTER-ZONE or UNW-INTRA-ZONE, depending on whether ω_h^n in M_1 is a zone delimiter.
- K_2 contains at least one zone delimiter: $K_2 = Z_0 :: \blacktriangleright\!\blacktriangleleft_{h_1}^{m_1} :: Z_1 :: \blacktriangleright\!\blacktriangleleft_{h_2}^{m_2} :: Z_2 :: \cdots :: \blacktriangleright\!\blacktriangleleft_{h_k}^{m_k} :: Z_k$ $(k \geq 1)$.
  By assumption of Lemma 1, the top-left SL configuration is well-typed, which, by the definition of the type system and the $\sim$ relation, implies that the bottom-left ZL configuration is also well-typed. It thus follows from Lemma 4 that $n \in \mathtt{Acc}_\infty\left(K_1 :: \omega_h^n :: K_2\right)$.
  Meanwhile, by the SL reduction step (top row) and rule RAISE, $n \notin \mathcal{L}(K_2)$. Thus, $n \notin \mathcal{L}(K_2)$.
  Combining the two results above, it must be that $n \in \mathtt{Acc}_{\min(m_1,m_2,\cdots,m_k)}\left(K_1 :: \omega_h^n :: Z_0\right)$. So by Lemma 2, $n < \min(m_1, m_2, \cdots, m_k)$. Since all zone delimiters in K_2 have labels greater than n, rule UNW-INTER-ZONE applies repeatedly, unwinding zones $\blacktriangleright\!\blacktriangleleft_{h_k}^{m_k} Z_k$ through $\blacktriangleright\!\blacktriangleleft_{h_1}^{m_1} Z_1$, until the zone walk reaches Z_0, resulting in the configuration $\langle K_1 :: \omega_h^n :: Z_0 \parallel \blacktriangleright\!\blacktriangleleft_{h_1}^{m_1} :: Z_1 :: \cdots :: \blacktriangleright\!\blacktriangleleft_{h_k}^{m_k} :: Z_k \parallel n \parallel v\rangle$. Then a single step takes this configuration to the bottom-right configuration, as in the base case. □

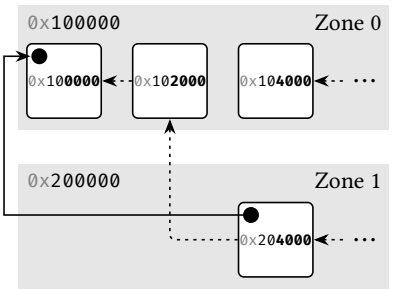
5 Implementation

We integrate the idea of virtual resumptions into the Lexa compiler [11]. Lexa is a simple functional language featuring lexical effect handlers. Its compiler translates a Lexa program into C, while using low-level stack-switching routines (packaged as a C library, called `StackTrek`) to implement effect handlers. The compiled C program is fed to LLVM for optimizations and code generation. `StackTrek` consists of x86 assembly functions for stack switching and macros `HANDLE`, `RAISE`, and `RESUME` that use those functions to install a handler, raise an effect, and resume a resumption. The compiler uses stack addresses to represent handler labels, enabling constant-time handler lookup. However, it cannot support multishot resumptions properly, as copying stack segments to new memory locations invalidates references to handlers installed on the copied stack.

5.1 Overview

A set of C functions implements the sMMU described in Section 2. This API for the sMMU manages the stack space using Linux's system calls for virtual memory management: zones are created by reserving large chunks of aligned virtual address space, and stack segments are allocated as

fixed-size blocks of 8 KB by committing the relevant memory range to physical memory. This allocation strategy allows the sMMU to maintain an ordered memory layout without unnecessary physical cost. The right diagram suggests the memory layout of zones and stack segments in them.



The Lexa compiler allocates a *header frame* at the base of each stack segment. A header frame contains information about the handler and points to the previous stack segment in the active stack (dashed arrows in the diagram). We use a flag in the header frame to indicate whether the stack segment is the start of a new zone. We call the first stack segment in each zone a *zone header*. A zone header points to the previous zone header in the active stack. In the diagram above, zone headers are marked with a black circle in the top-left corner, and the solid arrow stands for the pointer from a zone header to another.

The design laid out in Section 2.2 calls for a global counter to track the next available handler ID. We observe that a monotonically increasing counter is not necessary; it suffices that a freshly allocated handler ID is greater than any handler ID currently in use. So, as an optimization, the sMMU recycles handler IDs using a global bitmap that tracks the handler IDs in use.

The sMMU maintains another global bitmap to cache available zones, allowing it to reuse zones after they are freed. As a result, the sMMU operations usually avoid system calls.

Stack switching using the sMMU API. The sMMU manages stack allocation and address translation through its API: `get_stack`, `free_stack`, `translate_address`, and `dup_resumption`. The `StackTrek` library in the Lexa runtime is surgically modified (see Figure 11) to fully support multishot resumptions using this API.

`get_stack` takes the stack pointer `sp` as input, identifies the current stack segment's zone number and handler ID through `sp`, and allocates a new stack segment in the current zone with the next available handler ID. It is used on line 3, in the `HANDLE` macro, to allocate a new stack segment when installing a handler.

`free_stack` takes the stack pointer `sp`, identifies the stack segment through `sp`, and releases the associated memory. It is called in the `HANDLE` macro on line 6 to free the stack segment upon control leaving the lexical scope of the handler. `free_stack` updates the handler ID bitmap to recycle the handler ID. It also possibly updates the zone bitmap to recycle the zone memory.

`dup_resumption` takes the pointer to the resumption to be copied, walks through each stack segment in the resumption, and copies them into a fresh zone (line 13).

```
1  #define HANDLE(body, hdl_def)
2    char* new_sp = malloc(STACK_SIZE)
3                   get_stack(sp);
4    init_header(new_sp, hdl_def);
5    switch stack to new_sp and run body;
6    free_stack(new_sp);
7  #define RAISE(hdl, args)
8    hdl = translate_address(hdl);
9    adjust zone links;
10   char* saved_sp = hdl→saved_sp;
11   switch stack to saved_sp and run hdl→def(args);
12 #define RESUME(rsp, arg)
13   rsp = dup_resumption(rsp);
14   adjust zone links;
15   char* saved_sp = rsp→saved_sp;
16   switch stack to saved_sp and resume with arg;
```

Figure 11. Pseudocode implementation of `StackTrek` macros using the sMMU API. Code in red marks the main changes. A detailed account of stack switching with `StackTrek` can be found in Ma et al. [12].

`translate_address` takes the virtual handler reference, performs the address translation as described in Section 2.2, and returns the translated address. It is used by `StackTrek`'s `RAISE` macro on line 8.

5.2 Single-Shot and Last-Shot Optimization

A single-shot resumption is guaranteed not to be resumed again, so it is safe to destructively resume it by directly jumping into it. A multishot resumption can also be destructively resumed if it is the last time it will be resumed, which avoids the need for stack copying. Lexa has a `resume_final` construct that allows the programmer to indicate that a resumption is the last shot and thus can be resumed destructively. In principle, it is possible to use reference counting to determine whether to resume destructively; we leave this for future work as it is orthogonal to our focus.

A destructive resumption does not need to move the resumption stack to a new zone. Therefore, if the resumption lies in the same zone as the current execution point and has a larger handler ID, the runtime simply extends the current zone with the resumption stack. As a result, strictly single-shot programs will always stay in a single zone, so handler lookup remains constant-time. Programs that only use single-shot resumptions unlock further optimizations in LLVM. Empirical results in Section 6 show that these optimizations allow our new approach to incur minimal overhead compared to the existing Lexa implementation.

Resuming destructively introduces a complication: because the resumption stack is not physically moved to a new zone, it is possible to have multiple zone headers in the same zone. This means that the location of the nearest zone header cannot be determined using `sp` only. Instead, the sMMU uses a global pointer to track the nearest zone header during execution.

5.3 Optimzations for Abortive and Tail-Resumptive Handlers

A *tail-resumptive* handler invokes the captured resumption in the tail position, and an *abortive* handler does not resume the resumption at all. Lexa has optimizations for tail-resumptive and abortive handlers: a tail-resumptive handler is invoked in-place like a regular function call, whereas an abortive handler simply aborts the surrounding computation delimited by the header frame. Because there is no need to capture and restore a resumption, the compiler can avoid allocating a new stack segment and simply allocate the header frame in the middle of the current stack.

Our new runtime system supports these optimizations without extra effort. Even with the header frames located in the middle of a stack segment, the sMMU can still perform the same address translation as regular handlers to locate the correct handler instances.

5.4 Handler-Local Mutable State

Stack-allocated mutable state is challenging to implement in the presence of multishot resumptions: as each copy of the stack maintains its own version of the mutable state, care must be taken to ensure that mutations are performed on the correct version of the state.

Our approach can be adapted to support stack-allocated mutable state by employing the same zone-walking strategy to find the correct copy of the state. While we leave the full support to future work, our implementation supports a restricted form known as *handler-local* mutable state—state belonging to a handler instance and accessible through the handler reference. Since handler-local state is allocated in the handler's header frame, locating the correct version of the state is equivalent to locating the correct copy of the handler instance by walking zones.

5.5 Support for Simple Parallelism

As discussed in Section 2, our implementation strategy enables parallel execution of multiple copies of the same resumption. To support parallel execution, we extend Lexa with a parallel-let expression [10] that evaluates two sub-expressions in parallel and binds their results to two variables. Its syntax and operational semantics are provided in an appendix. Our implementation realizes parallel-let using pthreads and statically enforces that no effect escapes either of the parallel branches.

Since threads possess their own sets of zones, the global pointer to the nearest zone header is stored in thread-local storage, so that each thread holds its own value of the pointer. To avoid

different threads trying to use the same memory regions, each thread is assigned a distinct set of zones. A thread keeps track of the zones it owns using a thread-local bitmap and fetches new zones as necessary. A run-time error is raised if a thread tries to modify memory outside of its set of assigned zones, such as destructively resuming into another thread's zone.

5.6 Other Ways to Implement Virtual Resumptions

The implementation described in Section 5.1 represents one way to realize the idea of virtual resumptions. Other approaches are possible, with each approach coming with its own trade-offs.

In our approach, using virtual memory forces each stack segment to stay at a fixed address, which prevents relocation, such as allocating a larger stack when the original overflows. Zones also have fixed sizes, which allows locating zones using pointer arithmetic. While modern computers should have large enough virtual memory to accommodate our approach, there could also be cases where virtual memory is simply not a viable option.

We outline an alternative approach that uses a layer of indirection to allow more flexibility. The idea is to use a 2-dimensional array that holds pointers to the stack segments. One dimension identifies zones, while the other dimension identifies unique handler IDs. This indirection removes the restriction that a stack segment must be in a fixed location while still maintaining an ordering among handlers. Arrangements are needed to allow the stack segments to access their own indices in the array, such as by storing them in their header frames.

6 Evaluation

We assess the performance impact of integrating virtual resumptions into the Lexa compiler. We will refer to the original Lexa compiler as Base Lexa and the new compiler as Multi Lexa.

Base Lexa has only limited support for multishot resumptions: resumptions cannot span more than one stack segment, handler-local mutable state is disallowed, and multiple copies of the same resumption cannot be active simultaneously. These restrictions simplify the implementation of Base Lexa. In contrast, Multi Lexa removes the restrictions, fully supports multishot resumptions, but inevitably introduces some overhead due to the additional bookkeeping needed to maintain an organized memory layout. This motivates our first research question ***RQ1****: For benchmarks that Base Lexa can handle, does the additional bookkeeping in Multi Lexa introduce only small overhead compared to Base Lexa?*

Base Lexa and the most recent version of the Effekt compiler [13] both implement lexical effect handlers through direct manipulation of the call stack. However, both implementations are limited in their support for multishot resumptions: different copies of the same resumption cannot be active simultaneously. This restriction is fundamentally incompatible with running multishot resumptions in parallel. Multi Lexa removes this restriction, which motivates our second research question ***RQ2****: For benchmarks where multishot resumptions are embarrassingly parallel, does Multi Lexa unlock significant speedups by enabling the parallel execution of multishot resumptions?*

6.1 Overhead of Multi Lexa vs. Base Lexa

We use a community-maintained benchmark suite designed to evaluate the performance of effect handler implementations [2]. This suite primarily contains effect-heavy programs stressing the efficiency of handler invocation and resumption capture. On these benchmarks, Lexa delivers competitive results compared to other languages that support lexical effect handlers.

We gather additional benchmarks that heavily utilize multishot resumptions. These benchmarks are drawn from the literature on effect handlers. All except two are designed to run without handler-local state or extra handlers inside resumptions, so that Base Lexa can handle them. Notice that we could easily create more multishot benchmarks that require features not supported by Base Lexa or Effekt, but since the focus of comparison in Section 6.1 is on performance overhead

Table 1. Benchmark running times (milliseconds) measured for four implementations of lexical effect handlers: Base Lexa, Multi Lexa, Effekt, and Koka. The first group of benchmarks does not involve multishot resumptions, while the second group does. No parallelism among multishot resumptions is exploited by Multi Lexa in the runs reported in this table; the research question is answered in Table 2.

Benchmarks	**Base Lexa** [12]	**Multi Lexa**	**Overhead**	**Effekt** [13]	**Koka** [21]
Countdown	0	0	–	0	349
Fibonacci Recursive	508	583	14.8%	1847	1119
Product Early	99	99	0.0%	226	1765
Iterator	10	10	0.0%	9	487
Generator	856	861	0.6%	3265	7664
Resume Nontail	114	120	5.3%	99	1797
Parsing Dollars	271	271	0.0%	60	3038
Handler Sieve	474	484	2.1%	347	2165
Nqueens	283	362	27.9%	437	1714
Triples	202	215	6.4%	158	1684
Tree Explore	195	197	1.0%	269	250
Tree Explore BFS	99	74	−25.3%	186	125
Tree Generator	307	298	−2.9%	585	2085
Bellman	–	373	–	442	1297
Knapsack	461	513	11.3%	404	2639
Nim	319	304	−4.7%	392	695
Checkpointing	–	128	–	115	–

rather than expressiveness, we largely restrict ourselves to benchmarks that all implementations can handle.

With these benchmarks, we evaluate how Multi Lexa performs relative to Base Lexa, as well as to two other languages supporting lexical effect handlers—Effekt and Koka. Koka's *named handlers* [21] are a form of lexical handlers; we use them when implementing the benchmarks in Koka.

Zero- and single-shot benchmarks. The first set of benchmarks in Table 1 is the community-maintained benchmark suite [2] excluding those benchmarks that exercise multishot resumptions. Multi Lexa is able to achieve comparable results with Base Lexa, adding little to no overhead. For single-shot resumptions, Multi Lexa can perform the single-shot optimization discussed in Section 5.2, which obviates the need for zone walk and leads to further optimization opportunities for LLVM. The table also shows that Lexa is competitive with Effekt and Koka on these benchmarks.

Multishot benchmarks. The second set of benchmarks in Table 1 exercises multishot resumptions, comprising benchmarks from the community-maintained suite [2] and additional ones we gathered. These benchmarks cover a range of applications of multishot resumptions.

Nqueens performs backtracking search to find all solutions to the N-Queens problem. Multi Lexa incurs higher overhead on this benchmark compared to other benchmarks in this set, likely because Nqueens requires more resumption copying per raised effect, resulting in more stack segments being copied and freed throughout execution.

Tree Explore and Tree Explore BFS perform an exhaustive traversal over a tree structure in a depth-first and breadth-first manner, respectively. Interestingly, while Multi Lexa performs similarly to Base Lexa on Tree Explore, it gains a significant performance advantage on Tree Explore BFS. We suspect this is due to differences in memory management strategies: a breadth-first traversal using multishot resumptions requires storing continuations for all tree nodes of the same depth, which heavily stresses the efficiency of memory allocations and management.

Bellman implements the value iteration algorithm from Section 1. This benchmark installs two separate handlers for `Decision` and `Probability`. Base Lexa fails to support this benchmark due to some resumptions spanning two stack segments. Multi Lexa needs to do a single-step zone walk to locate the outer handler, but it is still competitive with Koka and Effekt on this benchmark.

Checkpointing is from Muhcu et al. [13]. It exercises both multishot resumptions and stack-allocated mutable state. While Base Lexa does not support such mutable state, Multi Lexa supports it using handler-local state discussed in Section 5. While Koka supports handler-local state to run this benchmark, it unfortunately cannot handle any reasonable input sizes due to stack overflow.

An appendix contains a detailed description of the other benchmarks and an analysis of Multi Lexa's performance on each benchmark. Another appendix contains scaling plots comparing the asymptotic performance of Multi Lexa with Base Lexa, Effekt, and Koka on each benchmark. Multi Lexa exhibits the same scaling behavior as Base Lexa, indicating that zone walk has only a constant overhead in these practical scenarios.

Overall, the results show that Multi Lexa incurs only moderate overhead on these multishot benchmarks relative to Base Lexa, while fully supporting multishot resumptions. In cases where the overhead is more noticeable (e.g., Nqueens and Knapsack), it is more than offset by parallel speedup, which we discuss next.

6.2 Speedup from Parallelizing Multishot Resumptions

Many applications of multishot resumptions are embarrassingly parallel, meaning that different copies of the same resumption can be executed in parallel without requiring synchronization or communication. Table 2 presents a selection of such benchmarks. Nqueens, Bellman, Knapsack, and Nim are straightforwardly parallelized versions of the benchmarks from Table 1, while Tree Explore Det is modified from Tree Explore to eliminate the need for synchronization.

Table 2. Multi Lexa running times (milliseconds) on sequential and parallel versions of several embarrassingly parallel multishot benchmarks. The last column shows the number of threads used in the parallel version, chosen based on the nature of the benchmark.

Benchmarks	Sequential	Parallel	# Threads
Nqueens	362	64	12
Tree Explore Det	197	86	4
Bellman	373	169	6
Knapsack	513	163	4
Nim	304	226	3

Nqueens, Bellman, and Nim are parallelized by resuming in parallel the multishot resumptions captured by the first invocation of the corresponding effects. Nqueens and Nim use a `Choose` effect to choose one of the 12 and three possible moves, respectively. Bellman involves two effects whose handlers use multishot resumptions: one chooses among three possible actions, and the other samples a distribution with sample size two, so we use six threads in total.

Tree Explore Det and Knapsack both use the `Choose` effect to choose between two branches. The branches at the top two levels are parallelized, so we use four threads in total.

Table 2 shows that parallelism significantly reduces the running time for all these embarrassingly parallel benchmarks, indicating that different copies of the same resumption can indeed be active simultaneously and thus executed in parallel. In contrast, neither Base Lexa nor Effekt admits such parallelism due to the restrictions they impose on multishot resumptions.

6.3 Summary

We conclude that the payoff of Multi Lexa, compared to prior implementation strategies, is twofold. In terms of expressiveness, Multi Lexa enables full support for multishot resumptions with minimal overhead on zero- and single-shot applications. In terms of performance, Multi Lexa may bring significant speedups by enabling parallel execution of multishot resumptions.

7 Related Work

Lexical effect handlers. Lexically scoped handlers were designed to recover abstraction safety while retaining the expressiveness of effect handlers [24; 23; 3]. Subsequent work on CPS translation for Effekt made these semantics practical for real languages [16; 14]. It was demonstrated in Lexa that handler lookup can be made constant-time using stack switching [12], and Effekt adopted the same technique in its LLVM back end [13].

Support for multishot resumptions. Some systems support effect handlers but not multishot resumptions, citing pragmatic reasons. For example, in OCaml 5, resuming a continuation more than once is a run-time error [17], a design choice motivated by the difficulty of reasoning about multishot continuations for both the programmer and the compiler.

For lexically scoped handlers, supporting multishot resumptions is technically challenging because stack copying invalidates references to handlers installed on the copied stack. Effekt [13] recently introduced support for multishot resumptions by using a global pointer to route handler references to the active resumption copy. However, this approach still detects and aborts "reentrant resumptions", precluding parallel execution of copies. Our virtual-resumptions technique removes this restriction while maintaining the fast lookup path expected of a stack-switching runtime, thereby enabling parallel execution of multishot resumptions.

Stack copying and address-space management. Implementation techniques for continuations have been studied extensively, and their trade-offs evaluated empirically [5; 8].

Beyond multishot resumptions, stack copying also arises when runtimes grow stacks by reallocation, as in Go [18] and OCaml [17]. Go's runtime reallocates a stack by copying it to a new memory location and patching all pointers into the stack to the new location. OCaml's runtime keeps a linked list of exception handlers on the stack, and these pointers are also patched during reallocation.

Effect handlers and parallelism. Xie et al. [22] extend an effect-handler calculus with a data-parallel `for` construct whose handler has a `traverse` clause to aggregate the effectful results of each iteration. Their goal is to parallelize independent loop bodies: each iteration runs in parallel and commits its contribution through the traversal. Our work is orthogonal. We target lexically scoped handlers in a stack-switching runtime and virtualize handler references so that a continuation may be resumed many times—even in parallel—while still retaining constant-time handler lookup in most cases.

8 Conclusion

Multishot resumptions can be implemented by stack copying, but stack copying is known to be at odds with stack-allocated resources—lexical effect handlers being a recent focus of interest. We resolve this tension with a new implementation strategy for lexical effect handlers that fully supports multishot continuations. The key insight is to give each copy of a continuation its own virtual view of handler identities. This virtualization is realized by a software-based memory management unit that performs efficient address translation, guided by the lexical scoping discipline of handlers. The accompanying formal model captures the essence of the approach and establishes its correctness. The implementation shows measurable speedups from enabling parallel multishot resumptions.

Acknowledgments

Data-Availability Statement

References

[1] Richard E. Bellman. 1957. *Dynamic Programming.* Princeton University Press.

[2] Bench [n.d.]. Effect handlers benchmarks suite. https://github.com/effect-handlers/effect-handlers-bench Accessed: 2025-10-06.

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

[4] 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. of the ACM on Programming Languages (PACMPL)* 6, OOPSLA1 (April 2022). https://doi.org/10.1145/3527320

[5] Will Clinger, Anne Hartheimer, and Eric Ost. 1988. Implementation strategies for continuations. In *Proc. of the ACM Conf. on LISP and Functional Programming (LFP)*. https://doi.org/10.1145/62678.62692

[6] Karl Crary, David Walker, and Greg Morrisett. 1999. Typed memory management in a calculus of capabilities. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/292540.292564

[7] Effekt [n.d.]. Effekt: A language with lexical effect handlers and lightweight effect polymorphism. https://effekt-lang.org Accessed: 2025-10-08.

[8] Kavon Farvardin and John Reppy. 2020. From folklore to fact: comparing implementations of stacks and continuations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385994

[9] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/512529.512563

[10] Robert H. Halstead. 1985. Multilisp: a language for concurrent symbolic computation. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 7, 4 (Oct. 1985). https://doi.org/10.1145/4472.4478

[11] Lexa [n.d.]. *The Lexa Programming Language*. https://github.com/lexa-lang/lexa

[12] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical effect handlers, directly. *Proc. of the ACM on Programming Languages (PACMPL)* 8, OOPSLA2 (2024). https://doi.org/10.1145/3689770

[13] Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. 2025. Multiple resumptions and local mutable state, directly. *Proc. of the ACM on Programming Languages (PACMPL)* 9, ICFP (Aug. 2025). https://doi.org/10.1145/3747529

[14] 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. of the ACM on Programming Languages (PACMPL)* 7, OOPSLA2 (Oct. 2023). https://doi.org/10.1145/3622831

[15] Gordon Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[16] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523710

[17] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454039

[18] Go Team. 2014. *Go 1.3 Release Notes: Stacks*. https://go.dev/doc/go1.3#stacks Accessed: 2025-11-13.

[19] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997). https://doi.org/10.1006/inco.1996.2613

[20] David Walker and Greg Morrisett. 2000. Alias types for recursive data structures. In *International Workshop on Types in Compilation*. https://doi.org/10.1007/3-540-45332-6_7

[21] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 6, OOPSLA2 (Oct. 2022). https://doi.org/10.1145/3563289

[22] Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. 2024. Parallel algebraic effect handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 8, ICFP (Aug. 2024). https://doi.org/10.1145/3674651

[23] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. of the ACM on Programming Languages (PACMPL)* 3, POPL (Jan. 2019). https://doi.org/10.1145/3290318

[24] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting blame for safe tunneled exceptions. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2908080.2908086

[25] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling bidirectional control flow. *Proc. of the ACM on Programming Languages (PACMPL)* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428207