Lexical Effect Handlers: Fast by Design, Correct by Proof

Cong Ma
University of Waterloo
Waterloo, Canada
cong.ma@uwaterloo.ca

Abstract

Lexical effect handlers offer both expressivity and strong reasoning principles, yet their practical adoption has been hindered by a lack of efficient implementations. As effect handlers gain mainstream traction, it becomes urgent to show that lexically scoped handlers can be made performant. My doctoral research addresses this need through the design and implementation of Lexa, a language and compiler that achieves state-of-the-art performance with lexically scoped handlers. Drawing on deep semantic insight, I have developed techniques that advance both implementation and theory. Lexa outperforms existing compilers, and all techniques are formally proved correct.

CCS Concepts: • **Software and its engineering → Compilers**; **Control structures**; **Correctness**; • **Theory of computation → Type structures**; **Control primitives**.

Keywords: lexical effect handlers, compiler correctness, Lexa.

ACM Reference Format:
Cong Ma. 2025. Lexical Effect Handlers: Fast by Design, Correct by Proof. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '25), October 12–18, 2025, Singapore, Singapore*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3758316.3762822

1 Motivation

Language design mistakes leave lasting scars—costing billions and shaping generations of software. Today, we stand at the brink of another such mistake. As a language and compiler designer, I see it as my responsibility to intervene before it becomes reality. This paper recounts my efforts to push past perceived limits and avoid repeating the errors of the past.

Effect handlers are a powerful language feature that enable expressive control flows, and mainstream language are racing to adopt them [7, 9]. They allow programmers to define custom control flow constructs, such as exceptions, async-await, and coroutines, in a modular way. Operationally, they resemble resumable exceptions: the handler captures the continuation from the point of the effect to the handler, allowing the program to resume it later.

However, recent research has revealed a modularity issue [11] with dynamically scoped effect handlers. The problem emerges in higher-order functions, where effects raised by function arguments may be accidentally handled by handlers installed in the enclosing higher-order function. The code below illustrates this issue: an application imports `plugin` and `framework`, and defines a wrapper `plugin'` that raises a `Logging` effect and intends to handle it inside the application. Now, if `framework` happens to install a `Logging` handler before calling the plugin, that handler will intercept the effect. This behavior may surprise the application developer and has been shown to violate the parametricity expected from higher-order functions. This is similar to the problem with dynamically scoped variables, where a variable defined in a higher-order function can be accessed by the function argument, leading to unintended consequences.

```
import plugin, framework
let plugin' = λx.plugin x; raise Logging(...)
handle
  framework(plugin')
with Logging = λx,k. print(x); resume k
```

In response to this issue, *lexically scoped handlers* [3, 11] have been proposed as a promising alternative. In this design, handlers are lexically scoped, and every effectful function is explicitly parameterized by a *handler label*. When a handler is installed, it generates a fresh label that is passed down to the effectful functions. The function can then raise effects that are routed to the appropriate handler via this label. Applied to the example, the `plugin'` is modified to explicitly accept a handler parameter. It is curried with the label associated with the `Logging` handler before being passed to the framework. This guarantees that the effect raised by the plugin is handled by the application's `Logging` handler, rather than the framework's. It has been shown [2, 11] that this lexical scoping semantics restores strong reasoning principles while preserving the expressiveness of effect handlers.

Although lexically scoped handlers are attractive in theory, they have not been proved to be practical. Inefficiencies in the existing implementation of lexically scoped handlers

SPLASH Companion '25, Singapore, Singapore

ACM ISBN 979-8-4007-2141-0/25/10
https://doi.org/10.1145/3758316.3762822

have hindered their adoption in practice, despite their theoretical advantages. It would be a shame if language designers in the coming decade defaulted to dynamically scoped handlers solely for efficiency. It would be a shame if the language design community were on the verge of repeating the same misstep made with dynamically scoped variables. My doctoral research aims to influence this trajectory by demonstrating that lexically scoped handlers can, in fact, be implemented efficiently. Strong reasoning principles and high performance need not be at odds—effect handlers can achieve both.

2 Problem

Crafting a good compiler is the art of bridging two seemingly disparate worlds: the mathematical realm of the functional language and the physical reality of machine code. Existing compilers for effect handlers, daunted by their complex semantics, often take a shortcut by translating to a simpler functional language—typically using CPS [8] transformations—and then relying on an "oracle compiler" to generate machine code. This approach leaves a wide semantic gap between source and target, often resulting in inefficient code.

My research develops the Lexa compiler, which compiles the source language all the way to assembly, exploiting the semantic features of both source and target languages to improve efficiency. Despite the complexity of both source and target languages and the compilation itself, all compilation techniques are formally proved correct, with ongoing work to integrate them into certified compilers like CompCert.

3 Lexa Language and Compiler

My research produces the Lexa language and compiler [5, 6]. In this section, I will present the key techniques that enable Lexa to achieve state-of-the-art performance and expand the design space for effect handler implementations.

In Direct Lexa [6], I compile labels to memory addresses and leverage the efficiency of random memory access to reduce the cost of effect raising from $O(n)$ to $O(1)$. In Zero Lexa [5], I exploit the second-class nature of labels to achieve a novel zero-overhead implementation. The former improves the performance of effects that are raised frequently, while the latter works well for infrequent effects. The Lexa language allows programmers to choose the appropriate compilation strategy for each individual effect, enabling fine-grained tuning within the same program.

3.1 Direct Lexa

The semantics of lexically scoped handlers poses both challenges and opportunities for compilation. Unlike dynamically scoped handlers, the raise site of a lexical handler is given a label which *directly* refers to the handler. Our compiler exploits this convenience and compiles the labels to stack addresses where the handler is installed. When an effect is raised to a label, the control *directly* goes to the stack address specified by the label, eliminating the need for traversing the call stack to find the handler.

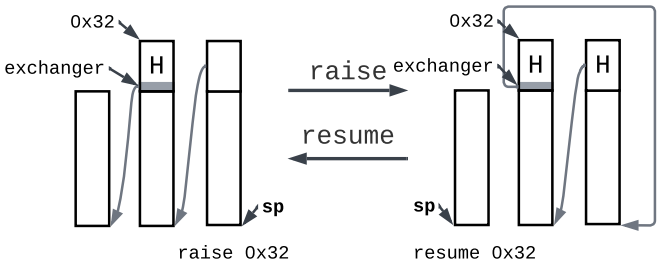

Figure 1. Stack switching in Lexa.

A key feature of effect handlers is their ability to reify and resume continuations as first-class values. It is important that Lexa supports this expressive feature without compromising performance. Lexa implements this via *stack switching*, which splits the runtime stack into segments, forming a chain of stacks. When an effect is raised, the stacks are cut at the place where the handler is installed. This is shown in Figure 1, where H represents the frame of a handler at address 0x32. The stack switching is carried out by simply swapping a special pointer, which we call *exchanger*, that was pointing to the previous stack with the current stack pointer sp. This simultaneously cuts the stack and reifies the continuation, which is now represented by the same address 0x32. To resume the continuation, another swap between the exchanger and sp restores the captured stack segments, allowing execution to continue from where it left off.

3.2 Zero Lexa

Widespread adoption of lexically scoped handlers would not be possible if this semantics altered the performance profile of existing code. Languages with exception handlers like C++ uphold the zero-overhead principle, which states "what you don't use, you don't pay for" [10]. Existing implementations for lexical handlers have so far failed to achieve this principle, as they all need to pass around the representation of the handler labels: [8] passes subregion evidences, and Direct Lexa passes memory addresses. These extra parameters add overhead, potentially slowing down existing programs.

Zero Lexa brings forth a breakthrough in this problem by introducing a type-directed compilation strategy that allows lexically scoped handlers to have the performance characteristics of their dynamically scoped counterparts, adhering to the zero-overhead principle. The core idea is as follows. During compilation, the compiler extracts handler provenance from the program's type structure and emits it as a static lookup table, which we call *hopper*. At run time, no handler labels are passed as parameters. Instead, when an effect is raised, a stackwalker traverses the call stack and consults the hopper at each frame to reconstruct the lexical scoping structure of the original source code. This enables the stackwalker to reconstruct the path the handler label would have taken

```
let fun0(ℓ2) = raise ℓ2
let fun1(ℓ0, ℓ1) = fun0(ℓ0)
```

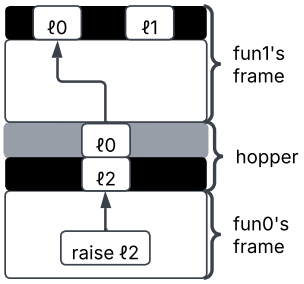

Figure 2. Zero Lexa stack and hopper.

had it been explicitly passed as a parameter. Because no handler label is threaded through the runtime, the mainline code remains untouched by the presence of handlers.

As an example, consider the code at the top of Figure 2. `fun0` takes a handler label ℓ_2 and raises to it, while `fun1` takes two labels, ℓ_0 and ℓ_1, and calls `fun0` with ℓ_0. The diagram below shows the call stack: black boxes denote function binders, and gray boxes show parameter instantiations. These together represent where the hopper conceptually resides on the stack; in the actual implementation, the hopper is a separate data structure indexed by return addresses. Note that label variables in the diagram are for illustration only and do not exist at run time.

When `fun0` raises to ℓ_2, the stackwalker begins by tracking ℓ_2. It consults the hopper to find its provenance and discovers that ℓ_2 was instantiated with ℓ_0 in `fun1`. The stackwalker then shifts to tracking ℓ_0 and continues upward, consulting the hopper at each step. This process continues until it reaches the frame where the label originated—i.e., where the handler was installed.

In a more complex program where the effects are abstracted over a higher-order function, it becomes less clear how the stackwalker should behave. As it traverses the frame of the higher-order function, the path to the handler is obscured by the abstraction, making it difficult to reconstruct the original lexical flow. To address this, the stackwalker maintains a dynamic state called a *clue*, which helps disambiguate among multiple potential paths. With the static hopper and dynamic clue, Zero Lexa faithfully simulates the semantics of the language that uses labels. See [5] for details.

4 Evaluation

We evaluate the Lexa compiler along two dimensions: correctness and performance.

4.1 Correctness

The Lexa compiler, comprising both Direct Lexa and Zero Lexa, is proved correct via a CompCert-style simulation proof [4]. For Direct Lexa, the proof establishes that the compiled assembly is semantically equivalent to the high-level source language. For Zero Lexa, the proof guarantees that the stackwalker accurately discovers the intended handler, as dictated by the source program that uses labels.

4.2 Performance

To evaluate Lexa's performance with effects that are raised frequently, we use a community-maintained benchmark suite [1] and compare against Effekt and Koka—both of which support lexically scoped handlers. The results show that Lexa consistently outperforms both across most benchmarks.

For infrequent effects, we introduce a new benchmark suite focused on exception-like patterns, where effects are raised rarely. Lexa again outperforms other systems in handling infrequent effects. We further compare Zero Lexa against Direct Lexa and validate our hypothesis: the cost of occasional stackwalking is outweighed by the benefit of avoiding label-passing overhead in mainline code.

References

[1] Bench [n. d.]. Effect handlers benchmarks suite. https://github.com/effect-handlers/effect-handlers-bench Accessed: 2025-03-01.

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 2, POPL (Jan. 2018). doi:10.1145/3158096

[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). doi:10.1145/3371116

[4] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal Automated Reasoning* 43, 4 (Dec. 2009). doi:10.1007/s10817-009-9155-4

[5] Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. 2025. Zero-Overhead Lexical Effect Handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 9, OOPSLA2 (Oct. 2025). doi:10.1145/3763177

[6] 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). doi:10.1145/3689770

[7] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 7, OOPSLA2 (Oct. 2023). doi:10.1145/3622814

[8] 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)*. doi:10.1145/3519939.3523710

[9] 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)*. doi:10.1145/3453483.3454039

[10] Bjarne Stroustrup. 2012. Foundations of C++. In *European Symp. on Programming (ESOP)*. doi:10.1007/978-3-642-28869-2_1

[11] 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). doi:10.1145/3290318

Received 2025-07-31; accepted 2025-08-16