Zero-Overhead Lexical Effect Handlers

CONG MA, University of Waterloo, Canada
ZHAOYI GE, University of Waterloo, Canada
MAX JUNG, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

Exception handlers—and effect handlers more generally—are language mechanisms for structured nonlocal control flow. A recent trend in language-design research has introduced lexically scoped handlers, which address a modularity problem with dynamic scoping. While dynamically scoped handlers allow zero-overhead implementations when no effects are raised, existing implementations of lexically scoped handlers require programs to pay a cost just for having handlers in the lexical context.

In this paper, we present a novel approach to implementing lexically scoped handlers of exceptional effects. It satisfies the zero-overhead principle—a property otherwise met by most modern compilers supporting dynamically scoped exception handlers. The key idea is a type-directed translation that emits information indicating how handlers come into the lexical context. This information guides the runtime in walking the stack to locate the right handler. Crucially, no reified lexical identifiers of handlers are needed, and mainline code is not slowed down by the presence of handlers in the program text.

We formalize the essential aspects of this compilation scheme and prove it correct. We integrate our approach into the Lexa language, allowing the compilation strategy to be customized for each declared effect based on its expected invocation rate. Empirical results suggest that the new Lexa compiler reduces run-time overhead in low-effect or no-effect scenarios while preserving competitive performance for effect-heavy workloads.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Control structures**; **Correctness**; *Semantics*; *Runtime environments*; • **Theory of computation** → **Type structures**; **Control primitives**; Functional constructs; *Operational semantics*.

Additional Key Words and Phrases: Lexa, exception handling, algebraic effects, compiler correctness.

ACM Reference Format:
Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. 2025. Zero-Overhead Lexical Effect Handlers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 399 (October 2025), 27 pages. https://doi.org/10.1145/3763177

1 Introduction

Exception handling is a common feature in modern programming languages, designed to help programmers manage abnormal or unusual run-time conditions. The idea is to allow control to be transferred in a nonlocal but structured manner to handler code, separating common-case code from the handling of exceptional conditions.

Effect handlers [Plotkin and Power, 2003; Plotkin and Pretnar, 2013] generalize exception handlers, effectively allowing programmers to define custom delimited control operators. They can express a wide range of control-flow patterns, including exceptions, coroutines, and cooperative multitasking. Mainstream languages such as OCaml, Scala, and WebAssembly have adopted or are considering adopting some forms of effect handlers.

Authors' Contact Information: Cong Ma, cong.ma@uwaterloo.ca; Zhaoyi Ge, zhaoyi.ge@uwaterloo.ca; Max Jung, max.jung@uwaterloo.ca; Yizhou Zhang, yizhou@uwaterloo.ca. All authors are affiliated with the David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada.

Ever since the inception of exception handling [Goodenough, 1975; MacLaren, 1977; Liskov and Snyder, 1979], language designers have always chosen to make exception handlers *dynamically scoped*: when an exception is raised, the dynamically closest enclosing handler is chosen to handle it. Designers of languages with effect handlers have largely followed this convention [Bauer and Pretnar, 2015; Leijen, 2017; Lindley et al., 2017; Sivaramakrishnan et al., 2021].

Lexically scoped handlers. There is a growing recognition that dynamically scoped handlers are not semantically well-behaved, for a reason similar to why dynamically scoped variables are not the go-to choice for language designers. It seems to have started with the observation that dynamically scoped exception handlers can accidentally catch exceptions not meant for them [Zhang et al., 2016]. It was further argued that this unintended interception of exceptions—and control effects more generally—is a symptom of a deeper modularity problem [Zhang and Myers, 2019]: the dynamic-scoping semantics compromises abstraction safety, leaking implementation details and obstructing local reasoning about effectful programs.

A wave of recent language-design efforts responds to the problems with dynamic scoping by making handlers *lexically scoped*. These designs include research languages such as Genus [Zhang et al., 2016], Effekt [Brachthäuser et al., 2020b], and Lexa [Ma et al., 2024]. Some form of lexically scoped handlers is also being considered for Scala. Even language designs that have mostly focused on dynamically scoped handlers, such as Koka and WebAssembly, have implemented or are considering support for lexically scoped handlers under the term *named handlers* [Xie et al., 2022; Phipps-Costin et al., 2023].

The core idea is that a handler serves as a lexically scoped capability. Only code that possesses the capability—either code within the handler's lexical scope or code that has received the capability from its caller—can raise effects to the handler. It has been shown that lexically scoped handlers support strong reasoning principles [Zhang and Myers, 2019; Biernacki et al., 2020] while preserving the expressive power of effect handlers. This development has sparked active research into compilation techniques for lexically scoped handlers [Zhang et al., 2016; Schuster et al., 2022; Xie et al., 2022; Müller et al., 2023; Ma et al., 2024].

> What you don't use, you don't pay for.
>
> — Stroustrup [1995, 2012]

The zero-overhead principle. We observe that all existing implementations of lexically scoped handlers share a common trait: they impose a run-time overhead on execution paths even when no effects are raised. A small overhead on mainline paths is acceptable in return for efficient handling of effects. However, there exist a broad class of programs that raise effects much less frequently than they perform effect-free computation. These programs still use handlers—in particular, exception handlers—to deal with unusual conditions, but their effect-free, mainline paths are expected to be traversed far more often than effect-raising paths. For such programs, the primary performance consideration for the compiler writer is to ensure that mainline paths are not slowed down by the presence of handlers.

This consideration is an instance of the *zero-overhead principle*: what you don't use, you don't pay for. In the context of exception handling, the term "zero-overhead exception handlers" means that code should not pay for exceptions unless they are actually raised. Notice that it does not mean that the raising and handling of exceptions should have no cost. On the contrary, it means just the opposite: it accepts a relatively high cost for exceptional paths, which are expected to be unusual, in exchange for minimal overhead on mainline paths, so that this trade-off leads to an overall performance gain.

> Normal case execution efficiency should not be impaired at all.
>
> — Atkinson, Liskov, and Scheifler [1978] and Liskov and Snyder [1979]

The zero-overhead principle has long shaped the design and implementation of exception handling. It dates back to CLU [Liskov et al., 1977], whose designers emphasized mainline-path efficiency as a key criterion for an implementation strategy. CLU's design methodology has influenced languages including C++ and Java—and more recently, WebAssembly [Wagner, 2017] and Python [Shannon, 2021]. In all these languages, exception handlers are *dynamically* scoped. Zero-overhead implementation strategies for dynamically scoped handlers are well understood, drawing on the seminal work of CLU.

However, zero-overhead *lexically* scoped handlers remain elusive. The semantics of lexically scoped handlers involves generating fresh labels that serve as identifiers of installed handlers and passing these labels down to code that can raise effects to the handlers [Zhang et al., 2016; Zhang and Myers, 2019; Biernacki et al., 2020]. This semantics suggests that implementations would need to represent these labels and pass the representations around at run time, which inevitably incurs some overhead even when no effects are raised. Indeed, all existing implementations do exactly what the semantics suggests: handlers are identified via pooled objects [Zhang et al., 2016], subregion evidence [Schuster et al., 2022; Müller et al., 2023], evidence vectors [Xie et al., 2022], or memory addresses [Ma et al., 2024], and therefore extra instructions are needed to propagate these identifiers.

It would be unfortunate if adopting lexically scoped handlers necessarily entailed a performance penalty on mainline execution, which could create a barrier to adoption despite the benefits of abstraction safety. Current implementation strategies, such as the approach of Ma et al. [2024], already keep the overhead on mainline execution low in practice. Nevertheless, compiler writers, loss-averse by nature, may hesitate to adopt lexically scoped handlers if they believe they are giving up efficiency for intangible gains that are not easily measured.

This work. We present a novel approach to compiling lexically scoped effect handlers that satisfies the zero-overhead principle. The key idea is a type-directed translation that enables the compiler to emit static information tracking the lexical provenance of handlers in scope. This information guides the runtime system in walking the stack to locate the handler for a raised effect.

The approach is complementary to, and compatible with, existing implementations of lexically scoped handlers: effects expected to be raised rarely can use the new zero-overhead implementation strategy, while effects expected to be raised more frequently can still use existing strategies. This flexibility allows the programmer to customize the implementation strategy based on the expected usage patterns of individual effects for optimal performance.

2 Main Ideas

Dynamically scoped handlers risk accidental handling of effects. Lexically scoped handlers address this modularity problem. To understand the risk of accidental handling, consider the example shown in Figure 1. It consists of four pieces: a library, a plugin, a framework, and an application, each developed without knowing the implementation details of the other pieces. The code in peach color pertains to lexical effect handlers and can be ignored for now.

- The library function `libFun` can raise the `Logging` effect to log messages. It is expected that the `Logging` effect will occur rarely under normal circumstances.
- Both `framework` and `plugin` call `libFun`, so they may further propagate the `Logging` effect raised by `libFun` to their callers.
- The `framework` function installs a `Logging` handler to handle `Logging` effects raised from its own call to `libFun`.
- The `framework` function is a higher-order function that receives an add-on from its caller and calls the add-on; it is oblivious to the effects that the add-on may raise.
- The `main` function passes `plugin` to `framework` as an add-on, so it is aware that `plugin` may raise `Logging` effects. As a result, `main` installs a `Logging` handler to handle `Logging` effects from `plugin`.

```
1  effect Logging =
2  | log : string → unit                      effect declaration
```

```
3  def libFun [logger: Logging] (...) =
4    ...
5    raise logger.log("...")
6    // logging effects are expected to occur rarely
7    ...                                      library code
```

```
8  def plugin [logger: Logging] (x: int) =
9    ...
10   libFun [logger] (...)
11   ...                                      plugin code
```

```
12 def framework [α] (addon: {α}int → unit) =
13   handle
14     ...
15     addon (...)
16     // framework is oblivious to effects raised by add-on
17     ...
18     libFun [fileLogger] (...)
19     ...
20   with fileLogger: Logging
21     // handler is intended for effects raised from line 18
22   | log(s) ⇒ logToFile(s)
                                              framework code
```

```
23 def main () =
24   handle
25     framework [consoleLogger] (plugin[consoleLogger])
26   with consoleLogger: Logging   // handler is intended for effects raised from plugin
27   | log(s) ⇒ logToConsole(s)                                            application code
```

Figure 1. A program consisting of a library, a plugin, a framework, and an application.

2.1 Handler Search Semantics and Implementation Strategies

Figure 2 shows the call stacks at the point when `libFun` raises a `Logging` effect during a call from `addon`. Each column depicts how a `Logging` handler is found using different semantics and compilation strategies. A triangle represents the `consoleLogger` handler installed in `main`, and a circle represents the `fileLogger` handler installed in `framework`. The gray arrows represent the searching behavior of the semantics or the compilation strategy.

For now, we focus on the first column, which shows the behavior of dynamically scoped handlers. When an effect is raised, the runtime searches for the nearest handler on the stack that can handle the effect. In this case, the handler installed in the `framework` frame handles the effect, and the message is logged to the file. In prior work [Zhang et al., 2016; Zhang and Myers, 2019; Biernacki et al., 2020], it is argued that this behavior is problematic; below, we explain why.

For modularity, the application should be able to swap a different implementation of the framework that has the same observable behavior, without noticing any difference. However, if the new framework does not call `libFun` and thus does not install a `Logging` handler, the effect raised from `addon` will be propagated over the framework and be handled in `main`—implementation details of the framework are leaked to its caller! This example demonstrates that dynamically scoped handlers threaten to leak implementation details of higher-order abstractions and also hinder modular reasoning.

Lexically scoped handlers address this modularity problem. When a handler is installed, a label is freshly generated, representing the newly created handler instance; effects can be raised to the handler instance only using this label. Function definitions can have extra bindings for labels, and function calls can take in labels as arguments. In Figure 1, both `libFun` and `plugin` take as input a label `logger` that represents a handler instance for the `Logging` effect. Labels can also be captured in closures. The `framework` function is parameterized by a *capability variable* α that abstracts over the possible labels that `addon` may capture—this polymorphism indicates that `framework` is oblivious to any effects possibly raised by `addon`. It might be useful to clarify the relationship between `plugin` and `addon`. One can view `addon` as a closure of `plugin` that captures the label `consoleLogger` from the environment at line 25. One can also view `addon` as a curried function of `plugin` that is partially applied with the label `consoleLogger`.

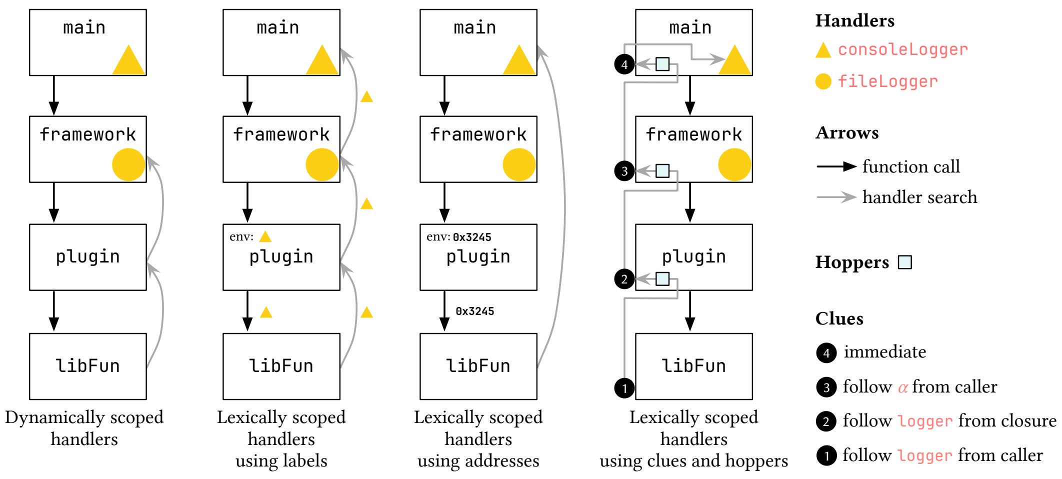

main
framework
plugin
libFun
Dynamically scoped handlers
env:
Lexically scoped handlers using labels
env: 0x3245
0x3245
Lexically scoped handlers using addresses
4
3
2
1
Lexically scoped handlers using clues and hoppers
Handlers
consoleLogger
fileLogger
Arrows
function call
handler search
Hoppers
Clues
immediate
follow α from caller
follow logger from closure
follow logger from caller

Figure 2. Stack diagrams illustrating different handler search semantics and implementation strategies.

The behavior of lexically scoped handlers for our example is shown in the second column of Figure 2. The function `plugin` captures `consoleLogger` in its closure environment. At run time, a label is freshly generated at line 24 upon entering the `handle` block in `main`, and it substitutes the `consoleLogger` in the `handle` block. We use a small triangle to represent this label in Figure 2. When `plugin` calls `libFun`, this label is passed to `libFun`, which then uses this label to raise the effect. The runtime searches for the nearest handler instance identified by this label, skipping over the handler installed in `framework` which has a different label, and eventually reaches `consoleLogger` installed in `main`. Readers should be convinced that regardless of the implementation details of the framework, the application has the guarantee that messages from `plugin` will be logged to the application's console rather than intercepted by the framework.

The third column in Figure 2 shows a representative, efficient implementation of lexically scoped handlers on a low-level machine [Ma et al., 2024]. Instead of generating fresh labels, the compiler simply uses the stack address of the handler as the label. When an effect is raised, the runtime directly jumps to the handler using the stack address. While the raising of effects is efficient in this implementation strategy, there is always an overhead associated with passing extra arguments even when effects are not raised.

While we use logging as an example of an effect, control effects in practice—especially in mainstream languages—most often appear as exceptions. Exception-handling code is common, but exceptions themselves are expected to be rare in normal circumstances. Thus, it is important to keep the overhead of exception handlers on mainline paths minimal. Even if the actual overhead is small with current implementation strategies, the mere violation of the zero-overhead principle can still discourage language designers from adopting lexically scoped handlers over dynamic scoping.

2.2 A Zero-Overhead Implementation Strategy

We contribute an implementation strategy for lexically scoped handlers that meets the zero-overhead principle. Unlike the existing approaches, this new strategy does not involve reifying or passing any representations of handler instances at run time. Rather, the information needed to locate the handler is spread across the call stack. When an effect is raised, the runtime system invokes a procedure called *stackwalker*. The stackwalker always carries a piece of information called *clue*. If the clue indicates that the stackwalker should further walk up the stack, the stackwalker will find in the caller's frame a transition function, called *hopper*, that maps the current clue to the next clue, which the stackwalker uses to continue the walk.

We now illustrate how the stackwalker works for our example, using the fourth column in Figure 2. For the purpose of presentation, in Figure 2, each clue is shown as a textual description of what the stackwalker should do next (see the **Clues** legend).

❶ When `libFun` raises the `Logging` effect, the initial clue tells the stackwalker to follow the `logger` label parameter of `libFun`, which is instantiated by `libFun`'s caller `plugin`.

❷ When the stackwalker reaches the call site of `libFun` in `plugin`, it discovers, through the hopper, that `plugin` calls `libFun` with a label named `logger` that comes from the closure environment of the partially applied `plugin`. Since the captured `logger` label must come from somewhere up the call stack, the stackwalker continues to walk up the stack.

❸ As the stackwalker reaches `framework`, it discovers, through the hopper, that the label captured in the callee `addon` is abstracted by a capability variable α. Now the stackwalker turns to follow α, which comes from the caller `main`. It skips over `fileLogger` installed in the frame of `framework`.

❹ Finally, as the stackwalker reaches `main`, it discovers, through the hopper, that the capability variable α of the callee `framework` is instantiated by the label `consoleLogger`, which represents the handler installed in the frame of `main`. As the handler for `consoleLogger` lives in the very same frame, the stackwalker reaches the handler implementation directly.

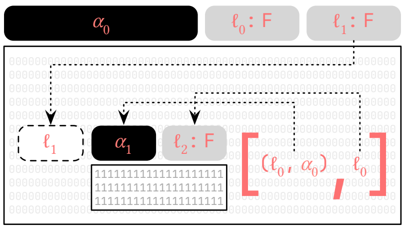
The hopper at a call site helps the stackwalker to transition from walking the callee's frame to walking the caller's frame. We have not yet explained what information hoppers store. In the rest of this section, we will use a more elaborate example in Figure 3, where functions are parameterized by multiple label variables (ℓ) and capability variables (α). The function `fun0` is parameterized over a capability variable α_0 and two label variables ℓ_0 and ℓ_1. The body of `fun0` consists of just a call to a nested function `fun1`. The function `fun1` is parameterized over a capability variable α_1 and a label variable ℓ_2. `fun1` takes in a function argument `g`, and `fun0` takes in a function argument `f` and passes it to `fun1`. The header of `fun1` also shows the label captured by `fun1` in curly braces $\{\ell_1\}$.

The right of Figure 3 depicts the definition of `fun0`. The box shaded with gray `0` corresponds to the body of `fun0`, while the box shaded with gray `1` corresponds to the body of `fun1`. The body of `fun0` consists of a single call to `fun1`, with the arguments $(\ell_0,\ \alpha_0)$ and ℓ_0 shown in the brackets. For each function body, the black boxes ⬛ above it are the capability variables parameterizing the function, and the gray boxes ░ are the label parameters. The white boxes with dashed borders ⬚ are the capture set of the function. `fun1`'s capture set is $\{\ell_1\}$. `fun0` has an empty capture set, because it is a top-level function. The dotted arrows ⇢ indicate the lexical provenance of the variables.

The main enabler of the zero-overhead implementation strategy is **a typing discipline that makes it explicit which of the three kinds of binders** (the legend at the bottom of Figure 3) **a handler is associated with.** This typing discipline provides a call site with sufficient knowledge—embedded in the hopper—about the provenance of a label or capability variable, so that it can use that knowledge to send the stackwalker up the call stack: just reverse the arrows in Figure 3! Hoppers can be computed at compile time, and the stackwalker only needs the return address of a call site to locate its hopper (see Section 5). Therefore, our implementation requires nothing to be put on the stack besides the return address, which is already there.

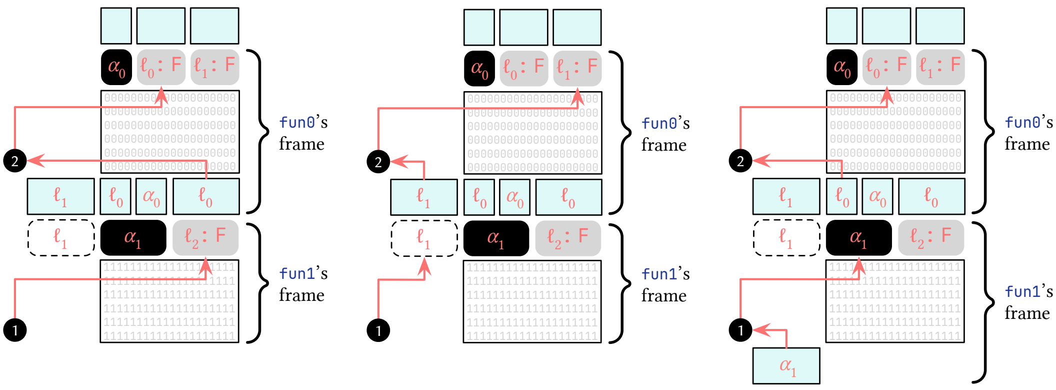
We use three stack snapshots (Figure 4) to illustrate how hoppers guide the stackwalker at run time. Each snapshot corresponds to a different effect raised from `fun1` or from further down the call stack. The stack frames are shaded with gray `0` or gray `1`, indicating whether the frame is for `fun0` or `fun1`. Between two stack frames are the three types of binders ⬚ ⬛ ░. Above each binder box is one or more light-blue boxes □—this is where the hopper *conceptually* resides on the stack.

```
1 effect F = unit → unit
2 def fun0 [α₀, ℓ₀: F, ℓ₁: F] (f: {ℓ₀, α₀}unit → unit) =
3   def fun1 {ℓ₁} [α₁, ℓ₂: F] (g: {α₁}unit → unit) =
4     raise ℓ₂ ()
5     raise ℓ₁ ()
6     g ()
7   fun1 [(ℓ₀, α₀), ℓ₀] (f)
```



◌ captured variable ● capability-variable binder ● label-variable binder ⇢ lexical provenance

Figure 3. Higher-order functions with assorted binding structures.



(a) line 4 (b) line 5 (c) line 6 (f's frame now shown)

Figure 4. Stack snapshots illustrating the behavior of the stackwalker when effects are raised from different points of the program in Figure 3.

The hopper records how each parameter of the callee is instantiated. It also records the lexical provenance of the captured variables in the protruding box.

- In snapshot (a), fun1 raises ℓ_2 (line 4 in Figure 3). The stackwalker starts by holding an initial clue ❶ that indicates the handler can be traced back to the label parameter ℓ_2 of fun1. So the stackwalker queries the hopper in the caller's frame. The hopper, given the clue ❶, returns a new clue ❷ that indicates the handler can be traced back to the label parameter ℓ_0 of fun0. The stackwalker then continues to walk up the stack.

- In snapshot (b), fun1 raises ℓ_1 (line 5 in Figure 3). ℓ_1 is a free variable captured by fun1. By our typing discipline, captured label or capability variables are reflected in a function's type (in curly braces), and the compiler includes their provenance information in the hopper emitted at the call site. Therefore, when the stackwalker queries the hopper and gives it the initial clue ❶, the hopper returns a clue ❷ that indicates the handler can be traced back to the label parameter ℓ_1 of fun0. The stackwalker then continues to walk up the stack.

- In snapshot (c), the effect is raised from further below fun1's frame, by the callee f of fun1 (line 6 in Figure 3). Recall that fun1 is a higher-order function that receives f as g and calls g.

  When the stackwalker queries the hopper at the call site of g, it gives the hopper a clue that indicates the handler comes from the capture set of g. Because the hopper records the lexical provenance of the capture set, it returns a clue ❶ that indicates the handler can be traced back to the capability variable α_1 parameterizing `fun1`.

The stackwalker then uses this clue ❶ to query the hopper at the call site of `fun1`. The hopper returns a clue ❷ that indicates the handler can be traced back to the label parameter ℓ_0 of `fun0`. How does the hopper know that it is ℓ_0 and not α_0 that the stackwalker should follow? It might be possible that the intended handler is further abstracted away by α_0, in which case the stackwalker should be directed to follow α_0. Section 2.3 addresses this possible ambiguity.

2.3 Nonambiguity Requirement on Capture Sets

To help the stackwalker determine the variable to follow when the call site is instantiated with more than one label or capability variables, we make the clue structure contain the name of the raised effect. Assume that the original raised effect is `F`. When the stackwalker queries the hopper at the call site of `fun1`, it determines that ℓ_0 has the matching effect name `F`, so it is the intended handler.

However, we still have not ruled out the possibility that the intended handler is in α_0. We prevent this possibility by eliminating the source of the ambiguity: we require that for every function *definition*, if its capture set is nonempty, then the capture set should contain either a single capability variable or labels of distinct effect names. With this restriction, as soon as the stackwalker finds a label with the matching effect name in the hopper, it need not look further. As Section 3 shows, this restriction is enforced by the type system and allows hoppers to always uniquely determine the handler to follow.

This nonambiguity requirement on the capture set of a function definition is mild. Current surface-language designs for lexically scoped handlers enforce a stronger condition [Zhang et al., 2016; Brachthäuser et al., 2020a]: if two effects of the same effect name are raised from the same function, then they must be handled by the same handler. This stronger condition allows the surface language to have a lightweight syntax that avoids explicit parameterization over capability and label variables. Our implementation strategy can be readily applied to such surface languages.

It is sometimes useful to allow multiple handler instances with the same effect name to coexist in the same scope [Xie et al., 2022]. Our implementation strategy supports this expressive power—with one caveat. First of all, our strategy fully supports functions parameterized by multiple handlers with the same effect name, as is shown by `fun0` in Figure 3, which is parameterized by ℓ_0 and ℓ_1 both having the effect name `F`.

The caveat is that the restriction on capture sets also applies to a code block being handled, potentially restricting expressiveness. Consider a computation C being handled by a handler H, as in the term `handle` C `with` H. The code block C needs to be typed like a function definition. This is because when an effect raised from within C is handled by H, the remaining computation in C may be reified as a continuation object, which behaves like a function and needs to be treated as such. Therefore, it is required that the labels in the capture set of C all have distinct effect names.

This restriction on C, however, is unnecessary if the handler H is tail-resumptive or abortive, because such handlers do not require reifying the continuation: the continuation either can be resumed in-place or does not resume at all. Since our implementation strategy is designed chiefly for exceptional effects, which are typically abortive, the practical impact of this restriction is minimal. This restriction can be further mitigated by allowing multiple handlers to be attached to the same code block at the same level of enclosure.

3 A Formal Model of the Zero-Overhead Compilation Scheme

In this section, we capture the essential aspects of the compilation scheme formally. The essence lies in a type-directed compilation between two languages: in the source language, handlers are identified by freshly generated labels passed down to effect-invocation sites, whereas in the target language, handlers are located purely by piecing together information left on the evaluation context. Everything else, to some extent, is an implementation detail.

To this end, we define two core languages, SL and TL. Both languages are based on the simply-typed lambda calculus, and we model their semantics as abstract machines. It is intentional that the source language is not high-level enough to be considered a surface language and that the target language is not low-level enough to be considered a machine language; we aim to distill the essence of the compilation scheme, by making the semantic gap between SL and TL only concerned with how handlers are located.

3.1 Source Language SL

SL programs use lexically scoped variables to identify handlers. In SL, every use of a handler explicitly refers to a *label variable* that stands for the handler. A surface language can offer a lighter-weight syntax by resolving implicit handler references to label variables bound in the lexical context, as in prior work [Zhang and Myers, 2019; Brachthäuser et al., 2020a].

Syntax. The syntax of SL is given in Figure 5. Auxiliary syntax needed for the operational semantics appears in Figure 6. We defer the type system to Section 3.3.

An overline denotes a sequence of (possibly empty) syntactic elements, with the empty sequence denoted by ϵ. For example, $\overline{\ell : F}$ denotes an ordered mapping from label variables to effect names. The i-th element of a sequence $\overline{\bullet}$ is denoted by a superscript $\bullet^{(i)}$. The size of a sequence $\overline{\bullet}$ is denoted by $|\overline{\bullet}|$. Capture-avoiding substitution is denoted by $\bullet[\bullet \mapsto \bullet]$.

Values include term-level variables, the unit value, and abstractions (i.e., function definitions). An abstraction $\left[\overline{\alpha}\,;\overline{\ell}\right](\overline{x}) \Rightarrow t$ can be parameterized by a sequence $\overline{\alpha}$ of *capability variables* and a sequence $\overline{\ell}$ of *label variables*. A *label* l substitutes for a label variable ℓ, and a *capability* T substitutes for a capability variable α. The parameterization over $\overline{\alpha}$ allows a higher-order function to be polymorphic over the capabilities required by its function arguments.

Expressions include function applications, `handle` expressions, the raising of effects, and the resuming of continuations. SL programs have undergone A-normalization [Flanagan et al., 1993] such that every subexpression, except the ones in `handle` and `with` clauses, is a value.

Function applications have the form $v_1\left[\overline{T};\overline{l}\right](\overline{v_2})$, where v_1 is the function, $\overline{T}$ is the sequence of capabilities instantiating the function's capability variables, $\overline{l}$ is the sequence of labels instantiating the label variables, and $\overline{v_2}$ is the sequence of arguments. A capability T is composed of a sequence of labels and at most one capability variable. That at most one capability variable is allowed in a capability is a simplification rather than a limitation; see the end of Section 3.3 for a discussion.

In a `handle` expression `handle` $[\ell : F] \Rightarrow t_1$ `with` $(x, k) \Rightarrow t_2$, the label variable ℓ is bound in t_1. The handler code t_2 takes two arguments: the payload x of the raised effect and the resumption k. A `raise` expression `raise` $l(v)$ raises an effect to the handler identified by label l, passing the value v as the payload.

A *term* `let` $x_1 = e_1$ `in` $\cdots$ `let` $x_n = e_n$ `in` v sequences the expressions e_1, ..., e_n, binding x_i to e_i in the rest of the term.

We make the standard simplifications that an effect signature contains exactly one effect operation and that an effect operation has exactly one argument.

Operational semantics. Figure 6 defines the operational semantics of SL as an abstract machine.

The syntax of labels l is extended with L. Metavariable L ranges over labels that are freshly generated at run time and thus do not appear in the program text.

A machine state M (a.k.a. *configuration*) is in one of three modes. In *normal mode*, the machine state $\langle E \parallel t\rangle$ consists of an evaluation context and a redex within that context. In *search mode*, the machine state $\langle E \parallel K \parallel L \parallel v\rangle$ consists of an evaluation context, a continuation under construction, effect name F label variable ℓ capability variable α term variable x, y, z, k

$$\begin{array}{rrcl}
\text{label} & l & ::= & \ell \\
\text{value} & v & ::= & x \mid () \mid \left[\overline{\alpha};\overline{\ell}\right](\overline{x}) \Rightarrow t \\
\text{capability} & T & ::= & \epsilon \mid \alpha \mid T, l \\
\text{expression} & e & ::= & v_1 \left[\overline{T};\overline{l}\right](\overline{v_2}) \mid \texttt{handle}\ [\ell : F] \Rightarrow t_1\ \texttt{with}\ (x,k) \Rightarrow t_2 \mid \texttt{raise}\ l(v) \mid \texttt{resume}\ v_1(v_2) \\
\text{term} & t & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ t
\end{array}$$

Figure 5. Syntax of SL.

$$\begin{array}{rrcl}
\text{label} & l & ::= & \cdots \mid L \\
\text{value} & v & ::= & \cdots \mid \texttt{cont}\ \#L^{(x,k)\Rightarrow t}K \\
\text{frame} & A & ::= & \texttt{let}\ x = \square\ \texttt{in}\ t \mid \texttt{let}\ x = \#L^{(y,k)\Rightarrow t_2}\square\ \texttt{in}\ t_1 \\
\text{evaluation contex} & E & ::= & \epsilon \mid E \cdot A \\
\text{continuation} & K & ::= & \epsilon \mid A \cdot K \\
\text{configuration} & M & ::= & \langle E \parallel t \rangle \mid \langle E \parallel K \parallel L \parallel v \rangle \mid \langle E \parallel K \parallel v \rangle
\end{array}$$

$$\begin{array}{rl}
\textsc{s-app} & \left\langle E \parallel \texttt{let}\ y = \left(\left[\overline{\alpha};\overline{\ell}\right](\overline{x}) \Rightarrow t_1\right)\left[\overline{T};\overline{L}\right](\overline{v})\ \texttt{in}\ t_2 \right\rangle \longrightarrow \\
& \left\langle E \cdot (\texttt{let}\ y = \square\ \texttt{in}\ t_2) \parallel t_1 \left[\overline{\alpha \mapsto T}, \overline{\ell \mapsto L}, \overline{x \mapsto v}\right] \right\rangle \\
\textsc{s-return} & \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel v \rangle \longrightarrow \langle E \parallel t[x \mapsto v] \rangle \\
\textsc{s-handle} & \langle E \parallel \texttt{let}\ x = (\texttt{handle}\ [\ell : F] \Rightarrow t_1\ \texttt{with}\ (y,k) \Rightarrow t_2)\ \texttt{in}\ t_3 \rangle \longrightarrow \\
& \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_3\right) \parallel t_1\,[\ell \mapsto L] \right\rangle \qquad \text{where } L \text{ is fresh} \\
\textsc{s-leave} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \parallel v \right\rangle \longrightarrow \langle E \parallel t_1\,[x \mapsto v] \rangle \\
\\
\textsc{s-raise} & \langle E \parallel \texttt{let}\ x = \texttt{raise}\ L(v)\ \texttt{in}\ t \rangle \longrightarrow \langle E \parallel \texttt{let}\ x = \square\ \texttt{in}\ t \parallel L \parallel v \rangle \\
\textsc{s-unw-let} & \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel K \parallel L \parallel v \rangle \longrightarrow \langle E \parallel (\texttt{let}\ x = \square\ \texttt{in}\ t) \cdot K \parallel L \parallel v \rangle \\
\textsc{s-unw-hdl} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L'^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \parallel K \parallel L \parallel v \right\rangle \longrightarrow \\
& \left\langle E \parallel \left(\texttt{let}\ x = \left(\#L'^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \cdot K \parallel L \parallel v \right\rangle \quad \text{where } L \neq L' \\
\textsc{s-found} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \parallel K \parallel L \parallel v \right\rangle \longrightarrow \\
& \left\langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t_1) \parallel t_2 \left[y \mapsto v, k \mapsto \texttt{cont}\ \#L^{(y,k)\Rightarrow t_2}K\right] \right\rangle \\
\\
\textsc{s-resume} & \left\langle E \parallel \texttt{let}\ x = \texttt{resume}\ \left(\texttt{cont}\ \#L^{(y,k)\Rightarrow t_2}K\right)(v)\ \texttt{in}\ t_1 \right\rangle \longrightarrow \\
& \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \parallel K \parallel v \right\rangle \\
\textsc{s-rew-let} & \langle E \parallel (\texttt{let}\ x = \square\ \texttt{in}\ t) \cdot K \parallel v \rangle \longrightarrow \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel K \parallel v \rangle \\
\textsc{s-rew-hdl} & \left\langle E \parallel \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \cdot K \parallel v \right\rangle \longrightarrow \\
& \left\langle E \cdot \left(\texttt{let}\ x = \left(\#L^{(y,k)\Rightarrow t_2}\square\right)\ \texttt{in}\ t_1\right) \parallel K \parallel v \right\rangle \\
\textsc{s-done} & \langle E \parallel \epsilon \parallel v \rangle \longrightarrow \langle E \parallel v \rangle
\end{array}$$

Figure 6. Operational semantics of SL as an abstract machine.

a label, and a payload. In *rewind mode*, the machine state $\langle E \parallel K \parallel v \rangle$ consists of an evaluation context, a continuation to be rewound, and a payload.

An evaluation context E is composed of a sequence of *frames* A.

- A frame can be a *let frame*, which takes the form `let` $x = \square$ `in` t.
- A frame can also be a *handler frame*, which takes the form `let` $x = \#L^{(y,k) \Rightarrow t_2} \square$ `in` t_1. A handler frame consists of a label L identifying the *handler instance* and the handler code $(y, k) \Rightarrow t_2$.

The notation $E[t]$ denotes the evaluation context E with the redex t placed in the hole of E.

A continuation K is similar to an evaluation context but is constructed inside-out rather than outside-in.

The syntax of values is extended with reified continuations `cont` $\#L^{(x,k) \Rightarrow t} K$. It represents a continuation K guarded by a handler frame. SL uses the standard deep-handler semantics [Kammar et al., 2013], so all continuations begin with a handler frame.

Figure 6 shows the selected small-step operational semantics of SL abstract machine. The SL syntax is in A-normal form, so the order of evaluation is already indicated by the syntax; no structural rules are needed for locating the next redex. The rules S-APP, S-RETURN, S-LEAVE, and S-DONE are straightforward. The rules S-HANDLE, S-RAISE, and S-RESUME are standard for a language with lexical effect handlers.

The S-HANDLE rule installs a handler frame onto the evaluation context. The handler frame contains a freshly generated label L that identifies this newly installed handler instance. The term t_1 in the `handle` clause then becomes the redex to be evaluated next. This generativity of labels is seen in the semantics of previous calculi or language implementations supporting lexical handlers [Zhang et al., 2016; Zhang and Myers, 2019; Biernacki et al., 2020; Brachthäuser et al., 2020a; Xie et al., 2022; Ma et al., 2024].

The S-RAISE rule suspends the current computation and transfers control to a handler. The first operand of `raise` is a label L identifying the handler instance that should handle the effect. The second operand is the payload (i.e., the argument to the effect operation). The machine transitions from normal mode to search mode, searching for a handler frame matching L and constructing a continuation representing the suspended computation. Once the machine is in search mode, the rules S-UNW-LET and S-UNW-HDL take over to unwind the frames off the evaluation context and accumulate the frames into the continuation being constructed, until a handler frame that matches the label L is found, at which point the S-FOUND rule takes over.

In S-FOUND, the machine transitions to normal mode, and the handler code t_2 recorded in the handler frame becomes the redex to be evaluated next. The accumulated continuation substitutes for the free variable k in the handler code t_2, and the payload v substitutes for y.

The S-RESUME rule restores a captured continuation on the evaluation context. In S-RESUME, the machine transitions from normal mode to rewind mode, installing the handler frame found at the top of the continuation to the evaluation context. Once the machine is in rewind mode, rules S-REW-LET and S-REW-HDL move the frames off of the continuation and put them onto the evaluation context. When there are no more frames to rewind, the S-DONE rule takes over, and the machine transitions back to normal mode, with the payload in the redex position.

3.2 Target Language TL

The target language TL is defined in Figure 7.

Syntax. The syntax of TL values, expressions, and terms looks similar to that of SL. The main difference lies in that no label variables or capability variables are directly present in TL: abstractions are not parameterized by them, and the `handle` construct does not bind any label variables. Crucially, the `raise` construct does not reference any label variable but instead uses a *clue*.

$$
\begin{array}{rrcl}
\text{label index} & \ell & ::= & \infty \mid \hat{i} \\
\text{capability index} & \alpha & ::= & \infty \mid \mathring{i} \\
\text{value} & v & ::= & x \mid () \mid (\overline{x}) \Rightarrow t \mid \texttt{cont}\ \#^{(x,k)\Rightarrow t}K \\
\text{clue} & C & ::= & \langle \ell, F \rangle \mid \langle \alpha, F \rangle \\
\text{capability} & T & ::= & \epsilon \mid \alpha \mid T,\ \ell \\
\text{call-site metadata} & H & ::= & T_0;\ \overline{T};\ \overline{\ell : F} \\
\text{expression} & e & ::= & v\ (\overline{v})^H \mid \texttt{handle}^T\ t_1\ \texttt{with}\ (x,k) \Rightarrow t_2 \mid \texttt{raise}\ C(v) \mid \texttt{resume}^T\ v(v) \\
\text{term} & t & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ t \\
\\
\text{frame} & A & ::= & \texttt{let}\ x = \square\ \texttt{in}\ t \mid \texttt{let}\ x = \square^H\ \texttt{in}\ t \mid \texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t}\square\right)^T\ \texttt{in}\ t \\
\text{evaluation contex} & E & ::= & \epsilon \mid E \cdot A \\
\text{continuation} & K & ::= & \epsilon \mid A \cdot K \\
\text{configuration} & M & ::= & \langle E \parallel t \rangle \mid \langle E \parallel K \parallel C \parallel v \rangle \mid \langle E \parallel K \parallel v \rangle
\end{array}
$$

$$
\begin{array}{rl}
\textsc{t-app} & \left\langle E \parallel \texttt{let}\ x = ((\overline{y}) \Rightarrow t_1)\,(\overline{v})^H\ \texttt{in}\ t_2 \right\rangle \longrightarrow \left\langle E \cdot \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t_2\right) \parallel t_1\,[\overline{y \mapsto v}] \right\rangle \\
\textsc{t-ret} & \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel v \rangle \longrightarrow \langle E \parallel t[x \mapsto v] \rangle \\
\textsc{t-ret-app} & \left\langle E \cdot \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right) \parallel v \right\rangle \longrightarrow \langle E \parallel t[x \mapsto v] \rangle \\
\textsc{t-handle} & \left\langle E \parallel \texttt{let}\ x = \left(\texttt{handle}^T\ t_1\ \texttt{with}\ (y,k) \Rightarrow t_2\right)\ \texttt{in}\ t_3 \right\rangle \longrightarrow \\
 & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_3\right) \parallel t_1 \right\rangle \\
\textsc{t-leave} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \parallel v \right\rangle \longrightarrow \langle E \parallel t_1\,[x \mapsto v] \rangle \\
\\
\textsc{t-raise} & \langle E \parallel \texttt{let}\ x = \texttt{raise}\ C(v)\ \texttt{in}\ t \rangle \longrightarrow \langle E \parallel \texttt{let}\ x = \square\ \texttt{in}\ t \parallel C \parallel v \rangle \\
\textsc{t-unw-let} & \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel K \parallel C \parallel v \rangle \longrightarrow \langle E \parallel (\texttt{let}\ x = \square\ \texttt{in}\ t) \cdot K \parallel C \parallel v \rangle \\
\textsc{t-unw-app} & \left\langle E \cdot \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right) \parallel K \parallel C \parallel v \right\rangle \longrightarrow \left\langle E \parallel \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right) \cdot K \parallel \text{hopper}_H(C) \parallel v \right\rangle \\
\textsc{t-unw-hdl} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \parallel K \parallel C \parallel v \right\rangle \longrightarrow \\
 & \left\langle E \parallel \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \cdot K \parallel \text{hopper}_{T;\epsilon;\epsilon}(C) \parallel v \right\rangle \text{ where } C \neq \left\langle \hat{0}, F \right\rangle \text{ for any } F \\
\textsc{t-found} & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \parallel K \parallel \left\langle \hat{0}, F \right\rangle \parallel v \right\rangle \longrightarrow \\
 & \left\langle E \cdot \left(\texttt{let}\ x = \square^{T;\epsilon;\epsilon}\ \texttt{in}\ t_1\right) \parallel t_2 \left[y \mapsto v, k \mapsto \texttt{cont}\ \#^{(y,k)\Rightarrow t_2}K\right] \right\rangle \\
\\
\textsc{t-resume} & \left\langle E \parallel \texttt{let}\ x = \texttt{resume}^T\ \left(\texttt{cont}\ \#^{(y,k)\Rightarrow t_2}K\right)(v)\ \texttt{in}\ t_1 \right\rangle \longrightarrow \\
 & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \parallel K \parallel v \right\rangle \\
\textsc{t-rew-let} & \langle E \parallel (\texttt{let}\ x = \square\ \texttt{in}\ t) \cdot K \parallel v \rangle \longrightarrow \langle E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \parallel K \parallel v \rangle \\
\textsc{t-rew-app} & \left\langle E \parallel \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right) \cdot K \parallel v \right\rangle \longrightarrow \left\langle E \cdot \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right) \parallel K \parallel v \right\rangle \\
\textsc{t-rew-hdl} & \left\langle E \parallel \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \cdot K \parallel v \right\rangle \longrightarrow \\
 & \left\langle E \cdot \left(\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\square\right)^T\ \texttt{in}\ t_1\right) \parallel K \parallel v \right\rangle \\
\textsc{t-done} & \langle E \parallel \epsilon \parallel v \rangle \longrightarrow \langle E \parallel v \rangle
\end{array}
$$

Figure 7. Syntax and operational semantics of TL.

A clue consists of the name of the effect being raised—akin to the situation in a language with dynamically scoped handlers—and an index. The index is in one of three forms: $\hat{i}$, $\mathring{i}$, or ∞. Their usage will be discussed in Section 3.3. For now, readers can think of them as de Bruijn indices for the SL variables they correspond to.

The TL syntax of applications, `handle` expressions, and `resume` expressions are enriched with *call-site metadata* H or T, which is produced during the translation from SL to TL. Hoppers are defined in terms of call-site metadata. When an effect is raised, the runtime uses the hopper at each call site in the evaluation context to update clues until the right handler is found.

Operational semantics. The operational semantics of TL is given as an abstract machine. An evaluation context can be a let frame, a handler frame, or a *call frame*. Unlike in SL, handler frames in TL are not marked by labels, as no labels are present in TL. A call frame $\texttt{let}\ x = \Box^H\ \texttt{in}\ t$ carries the call-site metadata H. A handler frame $\texttt{let}\ x = \left(\#^{(y,k)\Rightarrow t_2}\Box\right)^T\ \texttt{in}\ t_1$ carries the metadata T.

Similar to SL, the abstract machine of TL also operates in three modes. The difference is that, in search mode, the machine state uses a clue C rather than a label L. The initial clue is provided by the `raise` expression, and it is updated as the machine traverses the evaluation context.

The rules T-APP, T-RET, T-RET-APP, T-HANDLE, and T-LEAVE govern evaluation in normal mode. The T-APP and T-HANDLE rules install a call frame and a handler frame, respectively, attaching the call-site metadata H or T to the new frame.

The T-RAISE rule steps a `raise` $C(v)$ expression, with the machine transitioning from normal mode to search mode. Once the machine enters search mode, the unwinding rules T-UNW-LET, T-UNW-HDL, and T-UNW-APP take over.

The T-UNW-APP rule skips over a call frame during handler search. It updates the clue C using the call-site metadata H. The new clue $\mathsf{hopper}_H(C)$ is given by a meta-level function $\mathsf{hopper}_H(\cdot)$ indexed by the call-site metadata H. The definition of $\mathsf{hopper}_H(\cdot)$ will be given in Section 3.3.

The T-FOUND rule is in effect when the immediately enclosing frame is a handler frame and the clue index is $\hat{0}$, meaning that the handler frame is the one that should handle the effect. If the clue index is not $\hat{0}$, then the T-UNW-HDL rule is applied to skip over the handler frame and continue searching, with the clue updated using $\mathsf{hopper}_{T;\epsilon;\epsilon}$.

The only puzzle that remains is how hopper_H is defined to generate a new clue from an old clue and the call-site metadata H. Section 3.3 explains it after first defining a type system for SL.

3.3 Type-Directed Translation from SL to TL

We have now seen the untyped operational semantics of SL and TL. Section 3.3 formally defines the type-directed translation from SL to TL.

SL type system. Figure 8 presents the type system of SL. A type can be the unit type, a function type, or a continuation type. Function types and continuation types are annotated with a capture set T, which stands for the capability or label variables captured by the function or continuation.

The typing rules shown in Figure 8 simultaneously perform type checking and translation. We explain type checking first. The rules ST-UNIT, ST-VAR, and ST-LET are straightforward.

Expressions are type-checked under four contexts: Δ (resp. Σ) for capability variables (resp. label variables) *parameterizing* the immediately enclosing abstraction, Θ for the capability variable or label variables *lexically captured* by the immediately enclosing abstraction, and Γ for term variables. The typing rules are implicitly parameterized by a global context $\mathbb{F}$ mapping effect names to effect signatures. The use of the three contexts Θ, Δ, and Σ makes it explicit with which of the three kinds of lexical binders (cf. the legend at the bottom of Figure 3) a handler in scope is associated.

In the ST-FUN rule, the function body t is typed under the capture set T, whose contents may come from any of the typing contexts Θ, Δ, and Σ. The typing rule requires that the captured

$$\tau ::= \texttt{unit} \mid \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau_1}) \to \tau_2 \mid \{T\}\,\texttt{cont}\ \tau_1 \to \tau_2$$

$$\Theta ::= \epsilon \mid \alpha \mid \overline{\ell:F} \qquad \Delta ::= \epsilon \mid \Delta,\ \alpha \qquad \Sigma ::= \epsilon \mid \Sigma,\ \ell:F \qquad \Gamma ::= \epsilon \mid \Gamma,\ x:\tau \qquad \mathbb{F} ::= \epsilon \mid \mathbb{F},\ F:\tau\to\tau$$

$$\boxed{\Theta \mid \Sigma \vdash \ell \ :\ F \rightsquigarrow \ell} \qquad\qquad \boxed{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \alpha}$$

$$\frac{\mathrm{index}_\Sigma(\ell) = i \qquad \Sigma(\ell) = F}{\Theta \mid \Sigma \vdash \ell \ :\ F \rightsquigarrow \hat{i}} \qquad \frac{\ell : F \in \Theta}{\Theta \mid \Sigma \vdash \ell \ :\ F \rightsquigarrow \infty} \qquad \frac{\mathrm{index}_\Delta(\alpha) = i}{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \dot{i}} \qquad \frac{\alpha \in \Theta}{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \infty}$$

$$\boxed{\Theta \mid \Delta \mid \Sigma \vdash T\ \texttt{as}\ \Theta'\ \ \texttt{unamb}} \qquad\qquad \boxed{\vdash \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau_1}) \to \tau_2\ \ \texttt{nonfriv}}$$

$$\Theta \mid \Delta \mid \Sigma \vdash \epsilon\ \texttt{as}\ \epsilon\ \texttt{unamb} \qquad \frac{\Theta \mid \Delta \vdash \alpha}{\Theta \mid \Delta \mid \Sigma \vdash \alpha\ \texttt{as}\ \alpha\ \texttt{unamb}}$$

$$\frac{\Theta \mid \Delta \mid \Sigma \vdash T\ \texttt{as}\ \overline{\ell':F'}\ \texttt{unamb} \qquad \Theta \mid \Sigma \vdash \ell\ :\ F \qquad F \notin \overline{F'}}{\Theta \mid \Delta \mid \Sigma \vdash T,\ \ell\ \texttt{as}\ \overline{\ell':F'},\ \ell : F\ \texttt{unamb}}$$

$$\frac{\begin{array}{l}\forall \alpha_0 \in \overline{\alpha}.\ \exists i. \\ \quad \overline{\tau_1}^{(i)} = \{T'\}\forall\left[\overline{\alpha'};\ \overline{\ell':F'}\right]\left(\overline{\tau_1'}\right) \to \tau_2' \\ \quad \wedge\ \alpha_0 \in T'\end{array}}{\vdash \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau_1}) \to \tau_2\ \ \texttt{nonfriv}}$$

$$\boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v\ :\ \tau \rightsquigarrow v} \qquad \boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash e\ :\ \tau \rightsquigarrow e} \qquad \boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash t\ :\ \tau \rightsquigarrow t}$$

ST-UNIT

$$\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash ()\ :\ \texttt{unit} \rightsquigarrow ()$$

ST-VAR

$$\frac{\Gamma(x) = \tau}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash x\ :\ \tau \rightsquigarrow x}$$

ST-LET

$$\frac{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash e\ :\ \tau_1 \rightsquigarrow e \qquad \Theta \mid \Delta \mid \Sigma \mid \Gamma, x\ :\ \tau_1 \vdash t\ :\ \tau_2 \rightsquigarrow t}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ t\ :\ \tau_2 \rightsquigarrow \texttt{let}\ x = e\ \texttt{in}\ t}$$

ST-FUN

$$\frac{\begin{array}{c}\Theta' \mid \overline{\alpha} \mid \overline{\ell:F} \mid \Gamma, \overline{x:\tau_1} \vdash t\ :\ \tau_2 \rightsquigarrow t \\ \Theta \mid \Delta \mid \Sigma \vdash T\ \texttt{as}\ \Theta'\ \ \texttt{unamb} \qquad \vdash \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau_1}) \to \tau_2\ \ \texttt{nonfriv}\end{array}}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \left[\overline{\alpha};\ \overline{\ell}\right](\overline{x}) \Rightarrow t\ :\ \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau_1}) \to \tau_2 \rightsquigarrow (\overline{x}) \Rightarrow t}$$

ST-APP

$$\frac{\begin{array}{c}\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1\ :\ \{T\}\forall\left[\overline{\alpha};\ \overline{\ell:F}\right](\overline{\tau}) \to \tau_1 \rightsquigarrow v_1 \qquad \forall i. \Theta \mid \Delta \mid \Sigma \vdash \overline{T_1}^{(i)} \qquad \forall i. \Theta \mid \Sigma \vdash \overline{\ell_1}^{(i)}\ :\ \overline{F}^{(i)} \\ \forall i. \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \overline{v_2}^{(i)}\ :\ \overline{\tau}^{(i)}\left[\overline{\ell \mapsto \ell_1}\right]\left[\overline{\alpha \mapsto T_1}\right] \rightsquigarrow \overline{v_2}^{(i)} \qquad \Theta \mid \Delta \mid \Sigma \vdash T;\ \overline{T_1};\ \overline{\ell_1 : F} \rightsquigarrow H\end{array}}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1\left[\overline{T_1};\overline{\ell_1}\right](\overline{v_2})\ :\ \tau_1\left[\overline{\ell \mapsto \ell_1}\right]\left[\overline{\alpha \mapsto T_1}\right] \rightsquigarrow v_1\,(\overline{v_2})^H}$$

ST-HANDLE

$$\frac{\begin{array}{c}\mathbb{F}(F) = (\tau_1) \to \tau_2 \qquad \Theta \mid \Delta \mid \Sigma \vdash T\ \texttt{as}\ \Theta'\ \ \texttt{unamb} \\ \Theta \mid \Delta \mid \Sigma \vdash \tau_{\mathrm{ans}} \qquad \Theta' \mid \epsilon \mid \ell\ :\ F \mid \Gamma \vdash t_1\ :\ \tau_{\mathrm{ans}} \rightsquigarrow t_1 \\ \Theta' \mid \epsilon \mid \epsilon \mid \Gamma,\ x : \tau_1,\ k : \{T\}\,\texttt{cont}\ \tau_2 \to \tau_{\mathrm{ans}} \vdash t_2\ :\ \tau_{\mathrm{ans}} \rightsquigarrow t_2 \qquad \Theta \mid \Delta \mid \Sigma \vdash T \rightsquigarrow T\end{array}}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \texttt{handle}\ \ [\ell : F] \Rightarrow t_1\ \texttt{with}\ \ (x,k) \Rightarrow t_2\ :\ \tau_{\mathrm{ans}} \rightsquigarrow \texttt{handle}^T\ t_1\ \texttt{with}\ \ (x,k) \Rightarrow t_2}$$

ST-RAISE

$$\frac{\Theta \mid \Sigma \vdash \ell\ :\ F \rightsquigarrow \ell \qquad \mathbb{F}(F) = \tau_1 \to \tau_2 \qquad \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v\ :\ \tau_1 \rightsquigarrow v}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \texttt{raise}\ \ell(v)\ :\ \tau_2 \rightsquigarrow \texttt{raise}\ \langle \ell, F \rangle\,(v)}$$

ST-RESUME

$$\frac{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1\ :\ \{T\}\,\texttt{cont}\ \tau_1 \to \tau_2 \rightsquigarrow v_1 \qquad \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_2\ :\ \tau_1 \rightsquigarrow v_2 \qquad \Theta \mid \Delta \mid \Sigma \vdash T \rightsquigarrow T}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \texttt{resume}\ v_1(v_2)\ :\ \tau_2 \rightsquigarrow \texttt{resume}^T\ v_1(v_2)}$$

Figure 8. The SL type system and the type-directed translation from SL to TL: selected rules.

capability T be *unambiguous*, through the judgment $\Theta \mid \Delta \mid \Sigma \vdash T \text{ as } \Theta' \text{ unamb}$, so that T can be used as the context Θ' of captured variables for typing the function body. Nonambiguity of T means that if it is not empty, it must be composed of either a single capability variable or a list of label variables with distinct effect names. As mentioned in Section 2.3, this is not a strong restriction, as existing surface languages already make stronger assumptions. The ST-FUN rule also requires that the function type be *nonfrivolous*, which means that each capability variable parameterizing this function must appear in the capture set of some function argument. This condition serves as a sanity check that the capability variables stand for capabilities needed by the function arguments.

The ST-APP rule is standard, typing the function v_1 and the arguments $\overline{v_2}$ under the same contexts.

In the ST-HANDLE rule, the `handle` expression can be viewed as defining two functions with the body t_1 and t_2—t_1 is parameterized by ℓ, and t_2 is parameterized by k and x—and then applying the first function. The subterms t_1 and t_2 capture a set T of variables coming from the contexts Θ, Δ, and Σ. This capture set T is required to be unambiguous so that it can be used as the context Θ' for typing t_1 and t_2.[1] The continuation parameter k of the handler has a continuation type that carries the same capture set T. The subterm t_1 is typed under a label context with a single label $\ell : F$, which serves as a lexical identifier of the handler. It is required that the label ℓ not appear in the answer type τ_{ans}, to prevent ℓ from escaping—a standard requirement in type-and-effect systems for lexically scoped handlers [Zhang and Myers, 2019; Biernacki et al., 2020] or lexical regions [Tofte and Talpin, 1997; Crary et al., 1999; Grossman et al., 2002] to ensure type safety.

The ST-RAISE and ST-RESUME rules are standard. In ST-RAISE, the effect signature of the effect being raised is used to type-check the payload v. In ST-RESUME, v_1 is required to have a continuation type whose input type matches the type of the payload v_2, and the `resume` expression is given the answer type of the continuation type.

The type system of SL is similar to those in prior works on lexically scoped handlers—with two key differences. First, SL uses two kinds of contexts to distinguish function parameters from captured variables. This distinction is crucial for the implementation of the stackwalker, but does not restrict nor relax the typability of SL programs. Second, SL imposes a nonambiguity condition on the capture set of a function definition and that of a handled computation. This restriction, in theory, reduces the expressiveness of SL programs. But as argued in Section 2.3, it appears to be of little practical concern. As the typing conditions of SL are a strengthening of those in prior type systems shown to be sound, we omit the proof of type soundness for SL.

TL call-site metadata. The TL program generated from a well-typed SL program uses call-site metadata H to track the lexical provenance of the label variables and capability variables the callee in SL has access to. A call-site metadata H is a three-tuple $T_0; \overline{T}; \overline{\ell : F}$, where T_0 is the "capture set" of the function being called, $\overline{T}$ is the "capabilities" instantiating the capability variables of the function, and $\overline{\ell : F}$ is the "labels" instantiating the label variables of the function. Notice that call-site metadata is a concept in TL, so they contain TL indices rather than SL labels or capabilities.

A TL index is in one of three forms: $\hat{i}$, $\mathring{i}$, or ∞. The index $\hat{i}$ corresponds to a label variable at de Bruijn index i. The index $\mathring{i}$ corresponds to a capability variable at de Bruijn index i. The index ∞ refers to either a label or capability variable in the capture set.

SL-to-TL translation. Figure 8 defines the type-directed translation from SL to TL, shown in orange. Rules of the forms $\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \ell$ and $\Theta \mid \Delta \vdash \alpha \rightsquigarrow \alpha$ translate SL variables to TL indices. If an SL variable is bound by the immediately enclosing abstraction (i.e., it is in Δ or Σ), it is translated to its de Bruijn index; otherwise, it is translated to ∞. Rules of the forms

[1]As discussed in Section 2.3, we can relax this nonambiguity requirement on the capture set if the handler is tail-resumptive or abortive. We omit the formalization of this relaxation.

$$\boxed{\mathsf{hopper}_H(C) = C'}$$

$$\mathsf{hopper}_{T_0;\,\overline{T};\,\overline{\ell : F}}\left(\left\langle \hat{i}, \overline{F}^{(i)} \right\rangle\right) = \left\langle \overline{\ell}^{(i)}, \overline{F}^{(i)} \right\rangle$$

$$\mathsf{hopper}_{T_0;\,\overline{T};\,\overline{\ell : F}}\left(\left\langle \mathring{i}, F' \right\rangle\right) = \begin{cases} \langle \ell', F' \rangle & \text{if there is a unique } \ell' \text{ such that } \ell' : F' \in \overline{T}^{(i)} \\ \langle \alpha, F' \rangle & \text{else if there is an } \alpha \text{ such that } \alpha \in \overline{T}^{(i)} \end{cases}$$

$$\mathsf{hopper}_{T_0;\,\overline{T};\,\overline{\ell : F}}(\langle \infty, F' \rangle) = \begin{cases} \langle \ell', F' \rangle & \text{if there is a unique } \ell' \text{ such that } \ell' : F' \in T_0 \\ \langle \alpha, F' \rangle & \text{else if there is an } \alpha \text{ such that } \alpha \in T_0 \end{cases}$$

Figure 9. Definition of $\mathsf{hopper}_H(C)$.

$\Theta \mid \Delta \mid \Sigma \vdash T \rightsquigarrow T$ and $\Theta \mid \Delta \mid \Sigma \vdash T_0;\, \overline{T_1};\, \bar{l} \rightsquigarrow H$ are defined structurally and omitted here for brevity.

The most interesting rules are ST-APP, ST-HANDLE, ST-RAISE, and ST-RESUME. In ST-APP, the call-site metadata H is computed from the capture set T and the instantiation $\left[\overline{T_1}; \overline{\ell_1}\right]$. In ST-HANDLE and in ST-RESUME, the call-site metadata T is computed using the captured capability T. In ST-RAISE, the label variable ℓ is translated to an initial clue consisting of a TL index and an effect name.

The hopper_H function. Figure 9 shows the definition of hopper_H. Recall that it is used by the operational semantics of TL to update clues in search mode (Figure 7). The function is indexed by the call-site metadata H, and it accepts a clue C and returns a new clue. The function is defined by a case analysis on the index component of the clue C, which is in one of the three forms: $\hat{i}$, $\mathring{i}$, or ∞.

If the index is $\hat{i}$, the handler responsible for handling the effect in SL must have been passed into the callee as a label instantiating one of the label variables. Therefore, as the search enters the caller's frame, the index in the clue is updated to be the TL index of the label argument.

If the index is $\mathring{i}$, the handler responsible for handling the effect in SL must have been abstracted away by a capability variable when the callee was invoked. So we check out the capability $\overline{T}^{(i)}$ instantiating the capability index $\mathring{i}$. If it contains a label index ℓ' matching the effect name in the clue, then ℓ' must be responsible for handling the effect. Otherwise, the desired handler must have been further abstracted away by another capability index α, so the clue is updated to reference α.

Notice that here we rely on the nonambiguity and nonfrivolity conditions required of an abstraction: they together guarantee that, at run time, the capture set of any function that is called must contain labels of distinct effect names. So as soon as a label index that matches the effect name is found, it is guaranteed that this label index identifies the desired handler, because there cannot be any other label index with the same effect name that exists in the instantiation $\overline{T}^{(i)}$ or in the further instantiation of any capability index in $\overline{T}^{(i)}$.

The hopper_H function is partial; it is undefined if there is ambiguity in how to resolve an index $\mathring{i}$, which may cause the evaluation of TL programs to get stuck. The simulation result in Section 4 guarantees that this cannot happen to a TL program generated from a well-typed SL program.

If the index is ∞, the handler responsible for handling the effect must have been captured by the callee. The first component T_0 of the call-site metadata H is exactly the variables captured by the callee, so hopper_H looks up T_0 to find the index of the variable responsible for handling the effect. The nonambiguity condition plays a similar role here as in the previous case.

Relaxing a syntactic restriction on capabilities. In both SL and TL, it is required that a capability T contain at most one capability variable or capability index. We now explain why we have this restriction in the first place and how to relax it. This restriction simplifies the definition

$$
\begin{array}{rrcl}
\text{value} & v & ::= & \cdots \mid \{T\} \left[\overline{\alpha};\ \overline{\ell : F}\right] (\overline{x}) \Rightarrow t \mid \texttt{cont}\ \{\overline{L_F}\} \# L_F^{(x,k)\Rightarrow t} K \\
\text{expression} & e & ::= & \cdots \mid v_1 \left[T; \overline{T}; \overline{l}\right] (\overline{v_2}) \mid \{T\}\texttt{handle}\ [\ell : F] \Rightarrow t_1\ \texttt{with}\ (x,k) \Rightarrow t_2 \mid \texttt{resume}\ v_1\ [T]\ (v_2) \\
\text{label} & l & ::= & \ell \mid L_F \\
\text{frame} & A & ::= & \cdots \mid \texttt{let}\ x = \left(\{\overline{L_F}\} \# L_F^{(y,k)\Rightarrow t_2} \square\right)\ \texttt{in}\ t_1
\end{array}
$$

Figure 10. Selected syntax of SL*. The difference between SL* and SL is highlighted.

of hopper$_H$, as it means that there is at most one capability variable we need to follow when there is not a label with the matching effect name immediately present. It is possible to relax this restriction and allow T to contain multiple capability variables. This can be done by generalizing clues to carry a list of indices, which allows the stackwalker to follow multiple capability variables during handler search. Any of them might abstract away the handler label we are looking for. We do not pursue this generalization in this work, as we have not encountered a situation where it is necessary.

4 Compiler Correctness

We now prove that the translation presented in Section 3.3 is correct by proving that it preserves semantics. We first present the theorem statements and then present selected rules of the simulation relation $\sim$ used to establish the result. The full definition of the simulation relation and the proof can be found in appendices available in the extended version [Ma et al., 2025b]. For visual aid, in this section, we use colors to distinguish between SL (blue) and TL (orange).

Theorem 1 (Simulation). *Given a well-typed SL configuration M and a TL configuration M such that $M \sim M$, if $M \longrightarrow M'$, then there exists a TL configuration M' such that $M \longrightarrow^* M'$ and $M' \sim M'$.*

The semantics-preservation result follows from Theorem 1.

Corollary 1 (Semantics Preservation). *If $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash t : \tau \rightsquigarrow t$, and $\langle \epsilon \parallel t \rangle \longrightarrow^* \langle \epsilon \parallel v \rangle$, then exists v such that $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash v : \tau \rightsquigarrow v$ and $\langle \epsilon \parallel t \rangle \longrightarrow^* \langle \epsilon \parallel v \rangle$.*

It states that if a closed, well-typed SL program terminates with a value, then its translation in TL also terminates and produces the expected result. The proof framework of Corollary 1 is standard and can be found in Leroy [2009]. Such a proof consists in establishing three conditions. (1) Condition on initial states: if $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash t : \tau \rightsquigarrow t$, then $\langle \epsilon \parallel t \rangle \sim \langle \epsilon \parallel t \rangle$. (2) Condition on final states: if $\langle \epsilon \parallel v \rangle \sim M$, then $M = \langle \epsilon \parallel v \rangle$, and $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash v : \tau \rightsquigarrow v$. (3) Simulation (Theorem 1). The first two conditions accept straightforward proofs. We focus on proving the third condition.

An enriched SL syntax. We carry out the simulation proof using a version of SL, which we call SL*, where values and expressions carry additional type-level annotations. The annotations facilitate the definition of the simulation relations. The extra annotations in SL* can be obtained simply from the typing of SL programs.

The differences between SL* and SL are highlighted in Figure 10. In this enriched syntax, abstractions and `handle` expressions are enriched with a capture set $\{T\}$, which in SL only exists as part of types. Continuation values are also enriched with a capture set; because continuations are created at run time, the capture set of a continuation is in the form of a list of run-time labels.

Applications and `resume` expressions are enriched with the capture set of the function and continuation, respectively. Additionally, a label L is enriched with its effect name, as in L_F. We will omit the effect name when it is not relevant to the presentation.

Next, we define the simulation relation between SL* and TL. We present selected rules.

Relational contexts. Most relations are defined under a pair of contexts: *promise* Π and *evidence* Ξ.

$$
\begin{array}{lrcl}
\text{promise} & \Pi & ::= & \tilde{\Theta};\ \tilde{\Delta};\ \tilde{\Sigma} \\
\text{evidence} & \Xi & ::= & \epsilon \mid \Xi,\ C \mapsto L
\end{array}
\qquad
\begin{array}{lrcl}
\text{label promise} & \tilde{\Sigma} & ::= & \epsilon \mid \tilde{\Sigma},\ \ell : F \sim \hat{i} \\
\text{capability promise} & \tilde{\Delta} & ::= & \epsilon \mid \tilde{\Delta},\ \alpha \sim \mathring{i} \\
\text{capture promise} & \tilde{\Theta} & ::= & \epsilon \mid \alpha \mid \overline{l : F}
\end{array}
$$

Promises relate TL indices to SL* labels and capabilities that come from an abstraction's parameters or capture set. A promise Π has three components. $\tilde{\Sigma}$, the label promise, relates TL label indices to SL* label variables. $\tilde{\Delta}$, the capability promise, relates TL capability indices to SL* capability variables. $\tilde{\Theta}$, the capture promise, is either an SL* capability or a list of SL* labels with their effect names. While $\tilde{\Sigma}$ and $\tilde{\Delta}$ may only contain SL* variables, $\tilde{\Theta}$ may contain both variables and run-time labels. We will use $\varnothing$ to denote a promise where all three components are empty.

Evidences are used to relate clues in TL to run-time labels in SL*. Having an evidence $C \mapsto L$ in the relational context means that the TL abstract machine in search mode carrying the clue C will find the handler corresponding to the SL* label L. Since evidences relate run-time labels, they are a more dynamic notion than promises.

Label and capability relations. $\boxed{l \underset{\Xi|\tilde{\Theta};\tilde{\Sigma}}{\overset{l}{\sim}} C}$ $\boxed{T \underset{\Xi|\Pi}{\overset{T}{\sim}} T}$ $\boxed{T_0;\ \overline{T};\ \bar{l} \underset{\Xi|\Pi}{\sim} H}$

The definitions of the label and capability relations are given in an appendix. Informally, an SL* label l is related to a TL clue C if either the promise Π or the evidence Ξ relates them. The relation between capabilities T and T, as well as that between $T_0;\ \overline{T};\ \bar{l}$ and call-site metadata H, are defined structurally. Recall that H in TL has a three-tuple structure matching $T_0;\ \overline{T};\ \bar{l}$ in SL.

Value, expression, and term relations. $\boxed{v \overset{\text{val}}{\sim} v}$ $\boxed{e \underset{\Xi|\Pi}{\overset{\text{expr}}{\sim}} e}$ $\boxed{t \underset{\Xi|\Pi}{\overset{\text{term}}{\sim}} t}$

$$
\text{promise}\left(T;\ \overline{\alpha};\ \overline{\ell : F}\right) = T;\ \left\{\overline{\alpha}^{(i)} \sim \mathring{i} \mid i\right\};\ \left\{\overline{\ell : F}^{(i)} \sim \hat{i} \,\middle|\, i\right\}
$$

$$
\text{evidence}\left(\overline{L_{0_{F_0}}};\overline{\overline{L_{1_{F_1}}}};\overline{L_{2_{F_2}}}\right) = \left\{\left\langle \infty, \overline{F_0}^{(i)} \right\rangle \mapsto \overline{L_0}^{(i)} \,\middle|\, i\right\} \cup \left\{\left\langle \mathring{i}, \overline{\overline{F_1}^{(j)}}^{(i)} \right\rangle \mapsto \overline{\overline{L_1}^{(j)}}^{(i)} \,\middle|\, i,j\right\} \cup \left\{\left\langle \hat{i}, \overline{F_2}^{(i)} \right\rangle \mapsto \overline{L_2}^{(i)} \,\middle|\, i\right\}
$$

$$
\text{R-V-FUN}\quad \frac{t \underset{\varnothing|\text{promise}\left(T;\ \overline{\alpha};\ \overline{\ell:F}\right)}{\overset{\text{term}}{\sim}} t}{\{T\}\left[\overline{\alpha};\ \overline{\ell : F}\right]\ (\overline{x}) \Rightarrow t \overset{\text{val}}{\sim} (\overline{x}) \Rightarrow t}
\qquad
\text{R-E-APP}\quad \frac{v_1 \overset{\text{val}}{\sim} v_1 \qquad \overline{v_2} \overset{\text{val}}{\sim} \overline{v_2} \qquad T_0;\overline{T};\bar{l} \underset{\Xi|\Pi}{\sim} H}{v_1\left[T_0;\overline{T};\bar{l}\right]\ (\overline{v_2}) \underset{\Xi|\Pi}{\overset{\text{expr}}{\sim}} (v_1(\overline{v_2}))^H}
$$

$$
\text{R-V-CONT}\quad \frac{t \underset{\Xi|\varnothing}{\overset{\text{term}}{\sim}} t \qquad K \underset{\Xi\cup\{\langle \hat{0},F\rangle \mapsto L\}}{\overset{\text{cont}}{\sim}} K \qquad \Xi = \texttt{evidence}\left(\overline{L'_{F'}};\epsilon;\epsilon\right)}{\texttt{cont}\ \left\{\overline{L'_{F'}}\right\}\#L_F^{(x,k)\Rightarrow t}K \overset{\text{val}}{\sim} \texttt{cont}\ \#^{(x,k)\Rightarrow t}K}
\qquad
\text{R-E-RAISE}\quad \frac{l \underset{\Xi|\tilde{\Theta};\tilde{\Sigma}}{\overset{l}{\sim}} C \qquad v \overset{\text{val}}{\sim} v}{\texttt{raise}\ l(v) \underset{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}}{\overset{\text{expr}}{\sim}} \texttt{raise}\ C(v)}
$$

R-V-FUN states that an SL* function is related to a TL function if the function bodies are related through the promise computed from the SL* function's parameters and capture set. It might be surprising that the evidence used to relate the function body is empty, as SL* run-time labels could have been substituted into the function body, which one might expect to be related through the evidence. Within the function body, such run-time labels are related through the promise, which will be fulfilled by the evidence available at the call site of the function.

R-E-APP states that an SL* application is related to a TL application if the functions v_1 and v_1 are related, the arguments $\overline{v_2}$ and $\overline{v_2}$ are related, and also $T_0; \overline{T}; \overline{l}$ and the TL call-site metadata H are related. The last premise effectively guarantees that all the capabilities that this function application needs are either promised to be related by Π or are already related by Ξ.

R-V-CONT states that an SL* continuation is related to a TL continuation if the handlers t and t are related through the evidence Ξ computed using the labels $\overline{L'}$ captured by the continuation. It also requires that the continuations K and K are related through the same evidence Ξ extended with an extra evidence for the label L. Notice that the handler bodies t, in contrast to the function bodies in R-V-FUN, are related entirely through the evidence and use an empty promise.

R-E-RAISE states that an SL* `raise` expression is related to a TL `raise` expression if the SL* label l and the TL clue C are related and also the payloads v and v are related.

These relations are also indexed by a context relating free term variables. We omit this standard context here for conciseness.

Evaluation-context relation. $\boxed{E \underset{\Xi}{\overset{\text{ctx}}{\sim}} E}$

$$\frac{\forall C \in \mathrm{dom}(\Xi').\ \Xi\left(\mathrm{hopper}_H(C)\right) = \Xi'(C)}{\mathrm{truthful}_{\Xi|H}(\Xi')}$$

$$\text{R-EC-APP}\ \frac{E \underset{\Xi}{\overset{\text{ctx}}{\sim}} E \qquad t \underset{\Xi|\emptyset}{\overset{\text{term}}{\sim}} t \qquad \mathrm{truthful}_{\Xi|H}(\Xi')}{E \cdot (\texttt{let}\ x = \square\ \texttt{in}\ t) \underset{\Xi'}{\overset{\text{ctx}}{\sim}} E \cdot \left(\texttt{let}\ x = \square^H\ \texttt{in}\ t\right)}$$

$$\text{R-EC-HANDLER}\ \frac{E \underset{\Xi}{\overset{\text{ctx}}{\sim}} E \qquad t_2 \underset{\Xi|\emptyset}{\overset{\text{term}}{\sim}} t_2 \qquad t_1 \underset{\Xi'|\emptyset}{\overset{\text{term}}{\sim}} t_1 \qquad \mathrm{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi') \qquad \Xi' = \texttt{evidence}\left(\overline{L'_{F'}}; \epsilon; \epsilon\right)}{E \cdot \left(\texttt{let}\ x = \left\{\overline{L'_{F'}}\right\} \# L_F^{(x,k)\Rightarrow t_1} \square\ \texttt{in}\ t_2\right) \underset{\Xi' \cup \{\langle \hat{0}, F \rangle \mapsto L\}}{\overset{\text{ctx}}{\sim}} E \cdot \left(\texttt{let}\ x = \left(\#^{(x,k)\Rightarrow t_1} \square\right)^T\ \texttt{in}\ t_2\right)}$$

The evaluation-context relation uses a judgement $\mathrm{truthful}_{\Xi|H}(\Xi')$, which means that all the evidence mappings in Ξ' can be found in Ξ with an extra indirection through hopper_H. In the expression and term relations, the evidence Ξ can be viewed as input. By contrast, in the evaluation-context relation, the evidence should be viewed as output, which will serve as the input for relating expressions and terms plugged into the evaluation context.

R-EC-APP states that if an evaluation context E in SL* is related to an evaluation context E in TL and yields an evidence Ξ, then the extended evaluation contexts with an extra call frame are related and yield an evidence Ξ', as long as Ξ' is truthful with respect to Ξ and the call-site metadata H on the frame.

R-EC-HANDLER states that if an evaluation context E in SL* is related to an evaluation context E in TL and yields an evidence Ξ, then the extended evaluation contexts with an extra handler frame are related and yield a new evidence made of a truthful Ξ' and an evidence for the newly introduced label L.

Simulation proof. The proof of Theorem 1 is by a case analysis on the SL* transition relation $M \longrightarrow M'$. The proof can be found in an appendix.

5 Implementation

We implemented our design for the Lexa programming language. Lexa is a functional language featuring lexical effect handlers. Its compiler uses segmented stacks to implement continuations, which allows it to capture and restore continuations efficiently [Ma et al., 2024]. Lexa uses stack addresses to represent labels, which makes handler search efficient. In fact, it obviates the need for handler search, since the stack address of the handler is passed down the call chain to the place where the effect is raised. However, this implementation strategy incurs a small overhead on mainline paths. The present work extends Lexa with a new implementation strategy that satisfies

	capture set	capability instantiation	label instantiation
call site in SL	ℓ_1	$(\ell_0,\ \alpha_0)$	ℓ_0
call site in TL	$\hat{0} : F_1$	$(\hat{1} : F_0,\ \mathring{0})$	$\hat{1} : F_0$

(a) Call-site information for line 7 in Figure 3.

	∞	$\mathring{0}$	$\hat{0}$
F_0	NA	$\langle \hat{1}, F_0 \rangle$	$\langle \hat{1}, F_0 \rangle$
F_1	$\langle \hat{0}, F_1 \rangle$	$\langle \mathring{0}, F_1 \rangle$	NA

(b) Tabular representation of the hopper function.

$$\overbrace{\overset{64}{\texttt{0x4312}}}^{\text{return address}} \quad \overbrace{\overset{8}{42}}^{\text{frame size}} \quad \overbrace{\langle \overset{2}{0},\overset{4}{0},\overset{4}{0} \rangle \langle \overset{2}{0},\overset{4}{1},\overset{4}{F_0} \rangle \langle \overset{2}{0},\overset{4}{1},\overset{4}{F_0} \rangle \langle \overset{2}{0},\overset{4}{0},\overset{4}{F_1} \rangle \langle \overset{2}{1},\overset{4}{0},\overset{4}{F_1} \rangle \langle \overset{2}{0},\overset{4}{0},\overset{4}{0} \rangle}^{\text{hopper function}} \quad \overbrace{\overset{1}{0}}^{\text{in-place invocation?}}$$

(c) Binary representation of the hopper function.

Figure 11. Compiling a hopper instance.

the zero-overhead principle. We refer to the original implementation strategy as *Direct Lexa*, and the new one as *Zero Lexa*.

Our extended Lexa language allows the programmer to declare effects as *exceptional*. Exceptional effects are expected to be raised rarely, so they are compiled using the Zero Lexa strategy, while non-exceptional effects are compiled using the Direct Lexa strategy. Importantly, effects implemented with different strategies can coexist in the same program.

The Lexa compiler is written in OCaml. It compiles Lexa programs to C code, which is then compiled and optimized by LLVM. In Lexa, raising effects and resuming continuations are efficient, implemented using handwritten assembly as stack-switching routines. Since Zero Lexa differs from Direct Lexa only in how the runtime locates handlers, it has a similar compilation pipeline.

5.1 Hopper Table

At the core of our formal model of the zero-overhead implementation strategy is the hopper functions, which guide handler search frame by frame. It may appear that evaluating hopper at run time could be expensive. However, notice that each hopper instance only depends on the typing information at compile time, so we pre-compute each hopper instance into a static lookup table that maps a clue to the next clue. Moreover, since each hopper instance is uniquely associated with a call site, we build a global lookup table that maps the return address of each call site to the corresponding hopper function. This global lookup table also stores the mapping from return addresses to the frame sizes, which is needed for walking the stack. The global lookup table is stored in the program's data section, completely hidden away from mainline execution paths.

We demonstrate how to compile a hopper instance into a lookup table, using the program in Figure 3. Specifically, we show how to compile the hopper table for the call site at line 7.

The call-site information at line 7 is shown in Figure 11(a). The hopper function at this call site can be represented as a table shown in Figure 11(b), where each cell shows the output clue for an input clue. Each row corresponds to a possible effect name in the input clue, and each column corresponds to a possible index in the input clue. Because `fun1` accepts one capability variable and one label variable, the table has three columns, including ∞ for effects raised to the handlers in the capture set. NA means the input clue is impossible.

We can flatten this hopper instance into bits. The global hopper table consists of entries that look like the one in Figure 11(c). Each entry consists of the return address associated with the call site, the frame size, and the bit-encoded hopper function. The number above each value represents how many bits are used to encode the value beneath. We assume that the program defines at most 16 effect signatures, and at any program point, there are at most 16 label or capability variables in scope. Thus, we use four bits each to encode the effect name and index, plus a two-bit number to indicate the type of index ($\hat{i}$, $\mathring{i}$, or ∞). Each entry also contains a one-bit flag to indicate whether the caller is invoking a tail-resumptive handler in-place; this flag is explained in Section 5.4.

5.2 Stackwalker

The stackwalker is implemented as a function that takes an initial clue from the effect `raise` site. When it arrives at a frame, it uses the return address at the frame to fetch a new clue from the global hopper table. It then uses the clue to either locate the handler at the current frame, or to jump to the next frame. The stackwalker continues this process until it finds the handler.

5.3 Multishot Resumptions

The Direct Lexa implementation strategy does not fully support multi-shot resumptions, and we briefly explain why here. Direct Lexa represents handler labels as stack addresses, and passes these memory addresses down the call chain as arguments, which are stored on the stack as function-local variables. If a resumption is resumed more than once, the stacks would have to be copied. These copies all contain the same stack addresses pointing to the handlers in the original stack, which might not be the right handler for a specific copy. There is no easy way to efficiently recover the right handler from the stale stack addresses.

With the new implementation strategy, no run-time representation of labels is needed, so multiple copies of the same resumption do not interfere. This allows Zero Lexa to support multishot resumptions without extra effort. Still, we do not view full support for multishot resumptions as a key advantage of Zero Lexa: applications requiring multishot resumptions are usually effect-heavy and thus do not benefit from the trade-off that Zero Lexa is designed for.

5.4 Tail-Resumptive Handlers

Compilers that feature effect handlers often support optimizations for tail-resumptive handlers. A tail-resumptive handler invokes the resumption as its final action. Since the resumption is not used in an interesting way, the handler can be invoked in-place like a regular function, eliminating the overhead of capturing and restoring the continuation. However, with our new design, it is tricky to get such an optimization right. After a tail-resumptive handler is invoked in-place, if it further raises effects, these effects must be handled in the context where the handler was originally installed, which may be higher up the call chain. If the stackwalker starts the walk from the current frame, it may find the wrong handler.

To address this issue, we modify the compiler in two places. First, when the program invokes a tail-resumptive handler in-place, the clue of this handler is pushed to the stack before making the call. Second, when a stackwalker reaches a frame, it checks the flag in the hopper entry to determine whether the caller is invoking a tail-resumptive handler. If so, it loads the previously stored clue from the stack and uses it to walk to the frame where the tail-resumptive handler was installed, before switching back to the clue it currently has. In general, it is possible that the stackwalker encounters another tail-resumptive call site before switching to the original clue; therefore, the stackwalker keeps track of a stack of clues, repeatedly pushing and popping clues as it encounters tail-resumptive call sites and install sites.

5.5 Separate Compilation

Zero Lexa is compatible with separate compilation. Each compilation unit can be compiled separately and produce its own hopper table. At link time, hopper tables from all compilation units can be merged, with code addresses in the hopper tables updated based on relocation information. We leave the implementation of this feature as future work.

6 Evaluation

Our goal in this section is to understand the performance characteristics of the new zero-overhead strategy as implemented in the Lexa compiler. This new Lexa compiler supports two different implementation strategies making different trade-offs. We call them Direct Lexa and Zero Lexa.

The Direct Lexa strategy, described in prior work [Ma et al., 2024], makes handler search and continuation capture efficient, but it incurs a small overhead even when effects are not raised, thus violating the zero-overhead principle. Zero Lexa gives up some efficiency in handler search, trading it for minimal overhead on mainline paths. We expect Zero Lexa to have a performance advantage for effects that are rarely raised.

A benchmark suite [Bench, [n.d.]] exists that has been employed in past evaluations of effect-handler implementations [Müller et al., 2023; Ma et al., 2024]. This suite targets effect-heavy workloads, stressing the efficiency of handler search and continuation capture. On these benchmarks, Lexa delivers state-of-the-art, competitive results, as the Lexa compiler uses the Direct Lexa strategy for effects that are not declared as exceptional. Crucially, integrating the zero-overhead strategy does not degrade the quality of code produced by the Lexa compiler for effect-heavy programs.

No benchmark suite exists that stresses the efficiency of mainline code. We fill this gap by gathering a new benchmark suite. These benchmarks are similar in size and complexity to the existing ones, but they raise effects infrequently. Most of them use exceptions to indicate illegal arguments or other exceptional conditions that are not expected to occur in normal circumstances. We compare how Direct Lexa and Zero Lexa perform in such *low-effect* or *no-effect* scenarios.

We include a case study involving cooperative multitasking. By varying the length of time slices, we control the frequency that effects are raised and compare the performance of the two strategies.

An appendix contains a detailed comparison of the new Lexa compiler with other systems supporting lexically scoped handlers—specifically, Effekt and Koka. Across both sets of benchmarks, Lexa demonstrates strong, competitive performance.

6.1 Effect-Heavy Benchmarks

The first set of benchmarks in Table 1 comes from the community-maintained benchmark suite [Bench, [n.d.]]. These benchmarks are effect-heavy programs designed to stress the efficiency of handler search and continuation capture. The Lexa compiler uses the Direct Lexa strategy for these benchmarks, as it is the default strategy. So the **Direct Lexa** column reflects the out-of-the-box performance of Lexa on these effect-heavy programs, while the **Zero Lexa** column shows the performance when the Zero Lexa strategy is forced.

As expected, the performance of Lexa degrades significantly if it is forced to use the Zero Lexa strategy for these benchmarks, due to the cost of stackwalking, which becomes a major overhead when effects are raised frequently.

With Direct Lexa, the Countdown and Iterator benchmarks are optimized to constants through aggressive inlining by LLVM. However, with Zero Lexa, such optimizations are not feasible because the stackwalking logic introduces complexity that prevents LLVM from performing similar inlining.

Parsing Dollars contains nested handlers. Zero Lexa performs poorly on this benchmark, as the stackwalker needs to walk over multiple stack frames to find the handler. In contrast, with Direct Lexa, finding the handler for a raised effect is a constant-time operation.

Handler Sieve uses a chain of tail-resumptive handlers. As discussed in Section 5.4, under Zero Lexa, additional complexity is required to allow tail-resumptive handlers to be invoked in-place. This complexity incurs a significant overhead.

Table 1. Two sets of benchmarks are used: the existing suite of effect-heavy benchmarks and a new suite of effect-infrequent benchmarks we curated. Lexa's default behavior is reflected in the **Direct Lexa** column and the **Zero Lexa** column, respectively, for these two sets of benchmarks. The last two columns show the hopper size as a percentage of the binary size and the stackwalking time as a percentage of the total running time. All experiments were conducted on a workstation with an AMD Ryzen 7 CPU (4.5 GHz).

Benchmarks	**Direct Lexa** (ms)	**Zero Lexa** (ms)	**Loss / Gain** Zero vs. Direct	**Zero Lexa Hopper Size**	**Zero Lexa Stackwalk Time**
Countdown	0	2488	−Inf	2.87%	96.9%
Fibonacci Recursive	507	506	0.2%	0.45%	0.0%
Product Early	95	99	−4.2%	3.54%	1.9%
Iterator	0	285	−Inf	2.87%	92.4%
Nqueens	280	403	−43.9%	2.79%	55.4%
Generator	846	1956	−131.2%	2.38%	55.7%
Tree Explore	197	220	−11.7%	1.91%	19.2%
Triples	203	319	−57.1%	4.33%	54.1%
Resume Nontail	119	177	−48.7%	2.05%	48.5%
Parsing Dollars	271	2622	−867.5%	6.07%	99.6%
Handler Sieve	475	3556855	−748711.6%	4.34%	100.0%
Catalan	344	301	12.5%	5.12%	0.0%
Bézout	680	671	1.3%	4.53%	0.0%
Golomb	651	563	13.5%	2.91%	0.0%
Hofstadter Q	576	559	3.0%	3.30%	0.0%
Karatsuba	804	702	12.7%	6.95%	0.0%
Ackermann	3975	3954	0.5%	2.51%	0.0%
Palindrome Partition	112	113	−0.9%	5.03%	0.0%
Lattice Path	1002	971	3.1%	2.50%	0.0%
Two Threads	153	140	8.5%	2.03%	0.5%

6.2 Effect-Infrequent Benchmarks

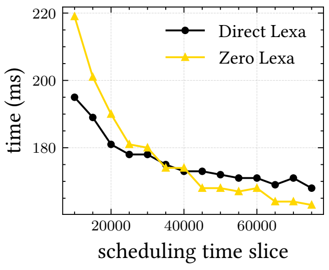
We now turn to the new benchmark set. It consists of programs and inputs that either produce no effects at run time or trigger them only rarely.

Catalan computes the n-th Catalan number, Bézout computes Bézout coefficients, Golomb computes the n-th number in the Golomb sequence, Hofstadter Q computes the n-th number of the Hofstadter Q sequence, and Karatsuba computes the product of two numbers using Karatsuba's algorithm. These programs use exceptions to indicate invalid input—for example, when the modulus of a modular division is not positive. Although exceptions do not occur at run time for the inputs we use, exception handlers still need to have run-time representations when Direct Lexa is used. Due to the recursive nature of the benchmark programs, passing these run-time representations of exception handlers around at run time is a major overhead. Zero Lexa does not incur this overhead and is hence faster.

Ackermann computes the Ackermann function, a simple recursive function with rapidly growing output. An exception handler is used to handle invalid input. Zero Lexa does not have a clear advantage over Direct Lexa on this benchmark, even though each function call in Direct Lexa uses an extra argument. We conjecture that this might be because the program is very simple, and the CPU can absorb the overhead of shuffling the extra argument around with out-of-order execution.

Palindrome Partition computes the number of possible palindromic partitions of a string. It raises an index-out-of-bounds exception when an illegal string index is accessed. Zero Lexa fails to gain any advantage over Direct Lexa on this benchmark, for reasons similar to Ackermann.

Lattice Path computes the number of 2D lattice paths between two points that avoid a third point. It contains a function that already takes six arguments, so the additional argument representing the exception handler in Direct Lexa introduces noticeable overhead. Zero Lexa does not require this additional argument and therefore outperforms Direct Lexa.



Two Threads runs two lightweight threads that cooperatively multitask on a shared OS thread. Each thread holds a counter representing its time slice and decrements the counter at the start of each recursive call. When the counter reaches zero, the thread raises a `Yield` effect, allowing the other thread to take over and run with a new time slice. The time slice is configurable and given as input to the program. When the time slice is large, the threads yield infrequently. In this case, the overhead of propagating the `Yield` handler is more significant than the overhead of handler search via stackwalking. Thus, in Table 1, we see that Zero Lexa is faster than Direct Lexa. However, when the time slice is small, the threads yield frequently to each other, and the cost of stackwalking dominates. This trade-off is illustrated in the figure above, which plots the running times of the two strategies against the time slice. As the time slice increases, the performance advantage shifts from Direct Lexa to Zero Lexa.

6.3 Hopper Size and Stackwalking Time

The second-to-last column of Table 1 shows the hopper size as a percentage of the binary size. The increase in binary size caused by Zero Lexa is modest; across all benchmarks, hoppers take up a single-digit percentage of the binary size.

The last column of Table 1 shows the stackwalking time as a percentage of the total running time. Stackwalking time varies significantly across benchmarks. In the effect-heavy benchmarks, it largely dominates the total running time. In the effect-infrequent benchmarks, it is negligible. These results confirm that the zero-overhead implementation strategy is best suited for low-effect or no-effect scenarios where effects are rarely raised. We expect future engineering work to reduce the cost of stackwalking further, but we leave these optimizations to future efforts.

7 Related Work

Almost all languages with an exception mechanism use dynamic scoping for exception handlers. CLU is one of the earliest languages to support exception handling [Atkinson et al., 1978; Liskov and Snyder, 1979]. The compiler generates exception tables that are consulted at run time to find the closest dynamically enclosing handler for an exception. Similar approaches are used in Java and most C++ implementations.

Dynamic scoping is flexible but potentially hazardous. Zhang et al. [2016] notice that dynamically scoped exception handlers can cause exceptions to be caught by accident. They address this problem in the Genus programming language [Zhang et al., 2015], with *tunneled exceptions*. In Genus, exceptions tunnel through program contexts oblivious to them. Tunneling, in its essence, is a form of lexically scoped handlers. It is based on the principle of local reasoning. The implementation works by a translation to Java exceptions and requires generating fresh identifiers for each handler instance at run time. This implementation strategy does not satisfy the zero-overhead principle.

What does it mean for an exception mechanism to prevent accidental handling? Zhang et al. [2019, 2020] suggest that the unintended interception of exceptions is a symptom of a deeper modularity problem that can be understood in analogy to a loss of *parametricity* [Wadler, 1989].

They construct logical-relations models for type-and-effect systems supporting *effect polymorphism* and prove that local reasoning principles are restored by lexically scoped effect handlers. In our SL, effect polymorphism exists in the form of parameterization over capability variables.

Biernacki et al. [2020] coined the term *lexically scoped effect handlers*. They investigate two semantics, an *open* semantics and a *generative* semantics and show that generativity is necessary when effect operations can be polymorphic. This generativity is seen in all existing implementations that support lexical handlers. Our work is the first to show that lexical handlers can be implemented without generating any form of reified representations of handlers.

Effekt supports lexical effect handlers, featuring lightweight effect polymorphism via the use of second-class values [Brachthäuser et al., 2020a], which is also used in Genus for lightweight exception polymorphism [Zhang et al., 2016]. The surface language does not require that handlers be explicitly named. A recent version of the Effekt compiler works by a translation to an intermediate language called System Ξ, where handlers are passed explicitly. System Ξ is further translated to a calculus Λ_{cap} with a region system [Müller et al., 2023]. This second translation is called lift inference, generating *subregion evidence* that determines how many handlers have to be jumped over until the right handler is found when an effect is raised. Λ_{cap} is subsequently translated to System F [Schuster et al., 2022]. This implementation strategy does not satisfy the zero-overhead principle, as subregion evidence has to be reified at run time as a sequence of function applications. A more recent version of Effekt moved away from this pipeline [Effekt Evolution, [n.d.]] and instead builds on Lexa's approach of compiling lexical effect handlers to stack switching [Ma et al., 2024].

Ma et al. [2024] observe that lexical handler search ought to run in constant time, yet prior implementations can incur costs akin to the search for dynamically scoped handlers. Lexa achieves constant-time handler search and resumption capture by compiling to stack switching, which improves asymptotic performance for effect-heavy workloads with deep stacks. This implementation strategy requires reifying handler instances as stack addresses. The present work extends the Lexa compiler by exploring a different trade-off aimed at low-frequency effects.

Koka is initially designed with dynamically scoped handlers [Leijen, 2017]. Xie et al. [2022] present a type-and-effect system for first-class named handlers in Koka, which are a form of lexically scoped handlers. This implementation of named handlers requires reifying handler instances as *evidence* information at run time and therefore does not satisfy the zero-overhead principle.

There is an ongoing effort to add some form of effect handlers to WebAssembly. The proposal is known as WasmFX [Phipps-Costin et al., 2023]. The design focuses on dynamically scoped handlers but also considers the possibility of named handlers. The work does not study implementation techniques for named handlers.

8 Conclusion

Lexically scoped handlers address a long-standing modularity problem with dynamic scoping, but existing implementations do not meet the zero-overhead principle for exceptional effects, which is otherwise satisfied by most modern compilers supporting dynamically scoped exception handlers. We have introduced a compilation strategy restoring this principle through a type-directed translation that emits information tracking the lexical provenance of handlers. The emitted information guides the runtime system in walking the stack to find the right handler for a raised effect. The payoff is reduced overhead on mainline paths, achieving parity with the performance profile of dynamically scoped exception handlers. Our approach complements existing techniques, allowing customization of compilation strategies for each declared effect to match their expected invocation frequencies. By identifying and removing a key barrier to adoption, we hope this work encourages language designers and implementers to embrace lexically scoped handlers for their modularity benefits.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any funding agency.

Data-Availability Statement

The artifact accompanying this paper is available [Ma et al., 2025a]. The latest release of the Lexa compiler can be found at the following link:

https://github.com/lexa-lang/lexa

References

Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. 1978. Aspects of implementing CLU. In *Proceedings of the 1978 Annual Conference (ACM '78)*. https://doi.org/10.1145/800127.804079

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015). https://doi.org/10.1016/j.jlamp.2014.02.001

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

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

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. of the ACM on Programming Languages (PACMPL)* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming (JFP)* 30 (March 2020). https://doi.org/10.1017/S0956796820000027

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

Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75, 5 (1972). https://doi.org/10.1016/1385-7258(72)90034-0

Effekt [n.d.]. Effekt: A language with lexical effect handlers and lightweight effect polymorphism. https://effekt-lang.org Accessed: 2025-07-01.

Effekt Evolution [n.d.]. A brief history of Effekt for fellow researchers. https://effekt-lang.org/evolution Accessed: 2025-07-01.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/155090.155113

John B. Goodenough. 1975. Exception handling: Issues and a proposed notation. *Comm. of the ACM* 18 (Dec. 1975). https://doi.org/10.1145/361227.361230

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

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/2500365.2500590

Koka [n.d.]. Koka: A functional language with effect types and handlers. https://koka-lang.github.io Accessed: 2025-07-01.

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3093333.3009872

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal Automated Reasoning* 43, 4 (Dec. 2009). https://doi.org/10.1007/s10817-009-9155-4

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

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3009837.3009897

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction mechanisms in CLU. *Comm. of the ACM* 20, 8 (Aug. 1977). https://doi.org/10.1145/359763.359789

Barbara H. Liskov and Alan Snyder. 1979. Exception handling in CLU. *IEEE Trans. Software Engineering* 5, 6 (Nov. 1979). https://doi.org/10.1109/TSE.1979.230191

Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. 2025a. *Zero-Overhead Lexical Effect Handlers (artifact).* https://doi.org/10.5281/zenodo.16928355

Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. 2025b. *Zero-Overhead Lexical Effect Handlers (Extended Version).* Technical Report CS-2025-04. School of Computer Science, University of Waterloo.

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

M. Donald MacLaren. 1977. Exception handling in PL/I. In *ACM Conf. on Language Design for Reliable Software.* https://doi.org/10.1145/800022.808316

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

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). https://doi.org/10.1145/3622814

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (Feb. 2003). https://doi.org/10.1023/A:1023064908962

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

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

Mark Shannon. 2021. *"Zero cost" exception handling.* https://bugs.python.org/issue40222

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

Bjarne Stroustrup. 1995. *The design and evolution of C++.* Addison-Wesley Publishing Co.

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

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

Philip Wadler. 1989. Theorems for free!. In *Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA).* https://doi.org/10.1145/99370.99404

Luke Wagner. 2017. *Should producers/consumers assume throwing is "rare" and, if so, can the spec note this?* https://github.com/WebAssembly/exception-handling/issues/19

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

Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, flexible object-oriented generics. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/2737924.2738008

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

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

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

Received 2025-03-26; accepted 2025-08-12