Lexical Effect Handlers, Directly

CONG MA, University of Waterloo, Canada
ZHAOYI GE, University of Waterloo, Canada
EDWARD LEE, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

Lexically scoping effect handlers is a language-design idea that equips algebraic effects with a modular semantics: it enables local-reasoning principles without giving up on the control-flow expressiveness that makes effect handlers powerful. However, we observe that existing implementations risk incurring costs akin to the run-time search for dynamically scoped handlers. This paper presents a compilation strategy for lexical effect handlers, adhering to the lexical scoping principle and targeting a language with low-level control over stack layout. Key aspects of this approach are formalized and proven correct. We embody the ideas in a language called Lexa: the Lexa compiler translates high-level effect handling to low-level stack switching. We evaluate the Lexa compiler on a set of benchmarks; the results suggest that it generates efficient code, reducing running-time complexity from quadratic to linear in some cases.

CCS Concepts: • **Software and its engineering → Compilers**; **Control structures**; *Functional languages*; *Assembly languages*; *Formal language definitions*; *Concurrent programming structures*; *Coroutines*; • **Theory of computation → Control primitives**; *Functional constructs*; *Operational semantics*.

Additional Key Words and Phrases: Algebraic effects, effect handlers, continuations, stack switching, compiler correctness, Lexa, Salt

ACM Reference Format:
Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 330 (October 2024), 29 pages. https://doi.org/10.1145/3689770

1 Introduction

Effect handlers [24, 25] are an attractive language abstraction for organizing control flow. They subsume a variety of linguistic features for nonlocal control flow such as exception handling and coroutine iterators. They provide a nice separation between the code that raises an effect and the code that handles it. They also afford a principled way to program continuations, with applications in cooperative multitasking, probabilistic programming, and more.

Conventionally, effect handlers are *dynamically scoped*: when an effect is raised, the dynamically closest enclosing handler is chosen to handle the effect. But recent work has found that dynamic scoping threatens abstraction safety and makes it hard to reason modularly about effectful programs [35, 3, 34].

In response to these challenges, *lexical effect handlers* have emerged as a promising design where effect handlers are lexically scoped [35, 34, 4, 7, 27]. A handler acts as a lexically scoped capability: only code holding the capability (i.e., code within the lexical scope of the handler or code that has

Authors' Contact Information: Cong Ma, cong.ma@uwaterloo.ca, School of Computer Science, University of Waterloo, Canada; Zhaoyi Ge, z33ge@uwaterloo.ca, School of Computer Science, University of Waterloo, Canada; Edward Lee, e45lee@uwaterloo.ca, School of Computer Science, University of Waterloo, Canada; Yizhou Zhang, yizhou@uwaterloo.ca, School of Computer Science, University of Waterloo, Canada.

ACM 2475-1421/2024/10-ART330
https://doi.org/10.1145/3689770

been passed the capability) is authorized to raise effects to the handler. It is shown that this lexical scoping semantics recovers strong reasoning principles while preserving the expressiveness of effect handlers.

This development in language design leads to the question for compiler writers: how might lexical effect handlers be implemented?

One way to implement lexical effect handlers is to piggyback on dynamically scoped handlers. This approach is taken by Genus [35] for compiling lexically scoped exception handlers to Java's exception handlers, which are dynamically scoped [12]. This approach necessarily inherits from Java the overhead of searching the stack for a matching handler at run time when an exception is raised.

Another approach is employed by the Effekt language, whose compiler also targets high-level languages. For example, one of Effekt's back ends performs a continuation-passing style (CPS) transformation to compile an intermediate language with lexical effect handlers to an ordinary functional language without handlers [27]. Before this CPS transformation can happen, the Effekt compiler first performs a *lift inference* to determine how many handlers have to be jumped over until the right handler is found [22]. It is believed that the lifting information computed by the lift inference saves the CPS-transformed code from the overhead of searching for handlers at run time. As we will see, this claim is not entirely accurate. The inferred lifting information *effectively* causes the generated code to walk the stack to locate the right handler.

In a certain sense, both approaches are rather roundabout. Lexical scoping is all about *static* reasoning. It is about knowing at compile time which handler in the lexical context will handle an effect raised at a given program point. Having to walk the stack at run time to locate the right handler seems at odds with the spirit of lexical scoping in theory—and can be a source of inefficiency in practice.

This insight begets the question: in a compiler targeting a low-level language that enables control over stack layout, can lexical effect handlers be implemented in a *direct* way that is faithful to the lexical scoping discipline, thus truly eliminating the need for the run-time search for handlers?

Our idea is a natural one. The low-level representation of a handler in scope is no different from that of any lexically scoped local variable. When an effect is raised, the control transfer target is *directly* available—no run-time search is needed to identify the handler. In addition, the suspended computation (aka resumption) is captured *directly* as stacks or stack frames without search.

Our idea is a natural one, also in the sense that it is hinted by the connection between lexical effect handlers and multi-prompt delimited control [13]. Prior work suggests a connection in semantics [34, 8, 36]. This work shows that the connection extends to implementation: our approach is reminiscent of native implementations of multi-prompt delimited control such as libmprompt [17], while incorporating optimizations not present in these implementations.

We proceed as follows. Section 2 reviews the design of lexical effect handlers and identifies a potential source of inefficiency in existing implementations. Section 3 defines two core languages: Lexa, an intermediate language with lexical effect handlers, and Salt, an assembly-like language with control over stack layout. Section 4 describes the translation from Lexa to Salt, and Section 5 describes optimizations for tail-resumptive and abortive handlers. Section 6 establishes the correctness of the translation. Section 7 describes the implementation of the Lexa compiler. Section 8 presents an empirical evaluation, comparing Lexa with other implementations of effect handlers.

2 Lexical Effect Handlers: A Tale of Two Schedulers

We use the example in Figure 1 to review the ideas of lexical effect handlers. The example, adapted from prior work [28], implements a cooperative lightweight multitasking scheduler. The example is written in an OCaml-like syntax. The type system is similar to OCaml, too, in that it does not track

```
1  (* library code *)
2  effect Process =
3  | yield : unit → unit
4  | fork : (Process → unit) → unit
5
6  effect Exn =
7  | throw : 'a
8
9  def driver q =
10   handle
11     val k = dequeue E q;
12     resume k ();
13     driver q
14   with E : Exn =
15   | throw k ⇒ log "all continuations done"
16
17 def spawn (f : Process → unit) q =
18   handle
19     f P
20   with P : Process =
21   | yield _ k ⇒ enqueue k q
22   | fork g k ⇒ enqueue k q; spawn g q
23
24 def scheduler (f : Process → unit) =
25   val q = mk_queue ();
26   spawn f q;
27   driver q
```

```
28 (* client code *)
29 def job P =
30   ...
31   raise P.yield ();
32   ...
33
34 effect Tick =
35 | tick : unit
36
37 def jobs P T n_jobs =
38   if n_jobs = 0 then
39     ()
40   else
41     raise T.tick;
42     raise P.fork job;
43     jobs P T (n_jobs - 1)
44
45 def main () =
46   val c = ref 0;
47   handle
48     scheduler (fun P → jobs P T 1000)
49   with T : Tick =
50   | tick k ⇒
51     c := !c + 1;
52     printf "forking job %d\n" !c;
53     resume k ()
```

Figure 1. Lightweight cooperative multitasking via lexical effect handlers.

effects [28]. It would be straightforward to define a type-and-effect system for lexical handlers; the problem is well studied [35, 34, 4, 7, 36] and orthogonal to the focus of this paper.

`Process` is an *effect signature* containing two *effect operations*, `yield` and `fork`. The scheduler uses an *effect handler* to interpret these operations.

The scheduler operates on a queue of continuations (aka resumptions). Initially, the queue is empty (line 25). The function `spawn` starts a computation `f` of the type `Process→unit`. As `f` may raise `Process` effects, `spawn` handles them by installing a handler for `Process` (lines 18–22).

Raising `yield` suspends the current job and hands control back to the scheduler. Raising `fork` additionally requests the scheduler to run a new job concurrently. The new job may itself raise `Process` effects. Specifically, when `yield` is raised, the remaining computation in the `handle` body is captured as a continuation `k`, which is entered into the scheduler queue, to be resumed later. When `fork` is raised, the handler additionally calls `spawn` recursively to run the new job.

After `spawn` returns, the scheduler calls `driver` to run the queued continuations (line 27). The function `driver` dequeues a continuation, resumes it, and recursively calls itself to run the next continuation (lines 10–15). If there is no more continuations to run (i.e., the queue is empty), an exception is thrown from the call to `dequeue`, and `driver` handles it by logging a message before exiting. Exn is the effect signature for exceptions, and throw is the effect operation for raising exceptions (lines 6–7).

We adopt the standard *deep handler* semantics [15]: when an effect is raised to a handler, the captured continuation contains the very handler in its outermost layer. At line 12, resuming a queued continuation may raise yield or fork. They are handled by the Process handler that is in the outermost layer of the continuation, namely the same handler that enqueued the continuation in the first place.

Handlers in this example are *lexical*. A handle expression binds a variable representing the handler in the handle body. For instance, the handle expression in spawn binds a handler named P (line 20), which is then used as an argument of f to handle the Process effects raised by f (line 19). A raise expression explicitly mentions the handler to which the effect is raised. For instance, the raise expression at line 42 specifies that the fork effect is raised to the handler named P, which is an argument of the function jobs. Although all uses of handlers are explicitly named in this example, prior work has shown that explicitly named handlers are not necessary in many cases; a lighter-weight syntax is possible by allowing omitted handler annotations to be resolved to handler bindings in the lexical context via a straightforward translation [34, 7].

The main function uses scheduler to run a jobs function, which further spawns 1000 jobs by raising fork effects. Each job may voluntarily yield to the scheduler (line 31).

In addition to running jobs, the programmer of the main function wants to keep count of how many times fork has been raised. This can be done by raising a Tick effect right before each fork in jobs (line 41). The main function handles Tick by incrementing a counter.

The costly tick. It is believed that Effekt [7, 22, 27], a language that most prominently features lexical effect handlers, does not require a run-time search for handlers when an effect is raised. So raising and handling effects should be cheap, comparable to function calls. As a result, it appears that the running time of the program should be proportional to the number of jobs.

We create a similar scheduler program in Effekt. Figure 2 shows how the running time scales. Surprisingly, we observe super-linearly increasing running time. Interestingly, if the Tick effect is turned off, linear scaling is restored. The seemingly innocuous Tick effect drastically changes the performance of the program!

Why are the Tick effects expensive for this scheduler implemented in Effekt? This experiment uses a back end of the Effekt compiler that translates control effects into a monadic framework for multi-prompt delimited control [9]. When an effect is raised, a sequence of prompt markers—each corresponding to a handler in the surrounding evaluation context—is traversed to find the right handler. In the scheduler program, a Tick effect is propagated through the entire call chain to the handler in main. The call chain, which is deep due to the recursive nature of driver, happens to contain as many Exn handlers as the recursion depth. So the asymptotic time complexity of handling a Tick effect is $O(n)$, where n is the number of jobs forked. Thus, the total running time of the program has $O(n^2)$ complexity.

Effekt also has a back end that first performs a *lift inference* [22] before a CPS translation to MLton. Unfortunately, this back end does not support the scheduler program due to a conflict between MLton's monomorphization requirement and the typed CPS translation. But even if the scheduler program could be compiled, it would not scale well. The lift inference generates a series of function applications that, at run time, effectively traverse the surrounding evaluation context to find the right handler for an effect.

Hence, both of these compilation strategies share a common inefficiency: although there is no run-time search of the *call stack* for handlers—there is not a call stack to begin with, as the compiler back ends do not need or manipulate call stacks—raising and handling an effect incurs a cost similar

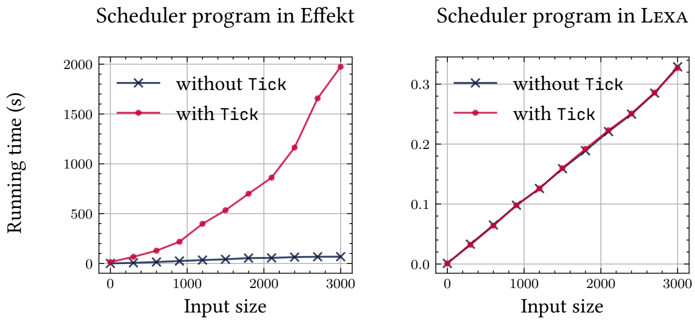


Figure 2. Scaling behaviors of the scheduler programs implemented in Effekt and in Lexa

to that required of a stack walk. We consider these compilation strategies as *lift-based*: they employ run-time computations to lift an effect out of the dynamic extents of handlers until the right handler is found.

Although quadratic scaling can be avoided by carefully adjusting the lexical scope of the exception handler so that only one exception handler is installed in the evaluation context at any time, it would be ideal if the programmer did not have to concern themself with such details. Furthermore, there exist useful programs for which deep stacks of handlers are unavoidable. For such programs, the scalability issue cannot be addressed merely by hand-tuning the scopes of handlers.

To lift, or not to lift? A lift-based approach is perhaps necessary for a compiler targeting a high-level language, but one may wonder if the run-time cost is avoidable for a compiler targeting a language that allows low-level control over the call stack. In a certain sense, lift goes against the grain of lexical handlers. The lift construct of Biernacki et al. [3] is a related language mechanism introduced to tame dynamically scoped handlers so that they do not compromise abstraction safety. The abstraction-safety problem does not arise in the context of lexical handlers [34], so lift ought to be unnecessary! This paper offers a strategy to compile lexical handlers without lift. As Figure 2 shows, the running time of the scheduler implemented in Lexa scales linearly, with or without `Tick` effects.

3 Two Core Languages: Lexa and Salt

We want to formalize the key aspects of the compilation of lexical effect handlers to a stack-based low-level language. Towards this goal, we define two core languages, Lexa and Salt. Sections 4–6 will define a translation from Lexa to Salt and prove its correctness. Lexa is a compiler intermediate language with support for lexical handlers. Salt is an assembly-like language with support for low-level control over memory and stack layout.

We use Lexa and Salt to capture the core ideas of how we compile lexical handlers; it is not a goal of this paper to formally verify a realistic compiler. Our compiler implementation described in Section 7 largely follows the formalism and supports additional features and optimizations. We have also implemented the formalism in Sections 3–5 faithfully in Racket, including the translation and an interpreter for the target assembly language; they can be found in the supplementary material.

3.1 Source Language: Lexa

Lexa programs use lexically scoped variables to identify handlers. Lexa is intended as a compiler intermediate language, where every use of a handler explicitly refers to the variable that binds the handler in the lexical context. A surface language can provide a lighter-weight syntax, by

$$\begin{array}{llcl}
\text{constant} & c & ::= & i \mid P \mid \texttt{ns} \\
\text{value} & v & ::= & x \mid c \\
\text{expression} & e & ::= & v \mid v_1 + v_2 \mid \texttt{newref}\ (v_0, \cdots, v_{n-1}) \mid \pi_i\ (v) \mid v_1\ [i] \leftarrow v_2 \mid v(v_1, \cdots, v_n) \mid \\
 & & & \texttt{handle}\ P_{\text{body}}\ \texttt{with}\ P_{\text{op}}\ \texttt{under}\ v_{\text{env}} \mid \texttt{raise}\ v_1\ v_2 \mid \texttt{resume}\ v_1\ v_2 \mid \texttt{exit}\ v \\
\text{term} & t & ::= & v\ \texttt{end} \mid \texttt{let}\ x = e\ \texttt{in}\ t \\
\text{program} & G & ::= & \texttt{letrec}\ P_1 = \lambda\overline{x_1}.\, t_1, \cdots, P_n = \lambda\overline{x_n}.\, t_n
\end{array}$$

Figure 3. Syntax of Lexa.

resolving uses of handlers to variables in scope, as described and formalized in prior work [34, 7]. To simplify the formalization, programs in Lexa are assumed to have undergone closure conversion and hoisting, so all functions, including those representing handlers, are closed and hoisted to the top level. Our compiler implementation does include these standard transformations.

Notably, because effect handlers in Lexa programs have a unified representation as functions, Lexa supports *bidirectional effects* [36], namely the ability for a handler to raise reverse-direction effects to the computation that raised the initial effect.

Similar to MultiCore OCaml [28], which also compiles to call stacks, it is not our goal to support handlers with multishot resumptions. Thus, in the Lexa formalism, a resumption can be resumed at most once. However, our compiler implementation does have limited support for multishot resumptions (Section 7.2).

As an intermediate language, Lexa is untyped, but it would be possible to extend Lexa with a type-and-effect system to ensure type safety, effect safety, and abstraction safety as in prior work [34, 36].

Syntax. Figure 3 presents the syntax of Lexa. *Expressions* are in A-normal form [26, 11], where every subexpression is a variable or a constant. *Terms* are sequences of expressions. A term `let` $x_1 = e_1$ `in` $\cdots$ `let` $x_n = e_n$ `in` v `end` sequences the expressions e_1, ..., e_n, binding x_i to e_i in the rest of the term. We will sometimes elide the trailing `end` for brevity. The notation $|t|$ denotes the number of `let` bindings within a term t.

Metavariable P ranges over *code labels*, which identify functions (including implementations of effect operations) in the program text.

A *value* is either a local variable x, an integer i, a code label P, or nonsense `ns`. Expressions and terms evaluate to values.

An *expression* takes one of the following forms: a value, an arithmetic operation, allocating a tuple, reading or writing a tuple, applying a function, installing a handler, raising an effect to an installed handler, and resuming a resumption.

In the expression `handle` P_{body} `with` P_{op} `under` v_{env}, P_{op} is the label of the handler code (i.e., the implementation of the effect operation), P_{body} is the label of the handled code, (i.e., the computation that may raise effects to the handler), and v_{env} provides the closure environment for both P_{op} and P_{body}. The environment is passed to the closures as an argument when they are called.

A *program* consists of a sequence of top-level, possibly mutually recursive functions, one of which is designated as the main function. We use the lambda notation $\lambda\overline{x}.\, t$ to denote a function, where an overline denotes a sequence of (possibly empty) syntactic elements. Effect handlers are defined as functions, with the last argument being the resumption. As a standard simplification, in the formalism, we assume that each handler has exactly one operation and that each operation has exactly one argument. Our implementation of the Lexa compiler is free of these restrictions.

$$\begin{array}{llll}
\text{constant} & c & ::= & \cdots \mid L \\
\text{term} & t & ::= & \cdots \mid \texttt{halt} \\
\text{code memory} & M & ::= & \{P_1 \mapsto \lambda\overline{x_1}.\, t_1, \cdots, P_n \mapsto \lambda\overline{x_2}.\, t_n\} \\
\text{heap value} & V & ::= & \langle v_0, \cdots, v_{n-1}\rangle \mid \texttt{cont}\ K \\
\text{heap} & H & ::= & \{L_1 \mapsto V_1, \cdots, L_n \mapsto V_n\} \\
\text{local environment} & E & ::= & [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n] \\
\text{frame} & F & ::= & (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \mid \#L^{P_{\text{op}}, L_{\text{env}}}\square \\
\text{evaluation context} & K & ::= & \square \mid K \cdot F \\
\text{configuration} & C & ::= & \langle M \parallel H \parallel K \parallel E \parallel t\rangle
\end{array}$$

Figure 4. Syntax of the Lexa abstract machine.

Operational semantics. We give an operational semantics to Lexa as an abstract machine. Figure 4 shows the extra syntax needed for defining the machine's semantics.

Metavariable L ranges over *data labels*, which are freshly generated at run time and thus do not appear in the program text. Data labels include ordinary *object labels*, freshly generated by evaluating `newref`, and *handler labels*, freshly generated by evaluating `handle`.

A machine state (i.e., *configuration*) consists of an immutable code memory, a mutable data memory (i.e., heap), an evaluation context, a local environment, and a redex.

- A code memory M maps code labels to functions.
- A heap H maps object labels to *heap values*. A heap value is either a tuple $\langle v_1, \ldots, v_n\rangle$ or a resumption `cont` K. Tuples can be used to represent closure environments, and resumptions are used to represent suspended computations.
- An evaluation context K is composed of a sequence of *frames* F.
  - A frame can be an *activation frame* $(E, \texttt{let}\ x = \square\ \texttt{in}\ t)$, which consists of a local environment E and a continuation that expects a value from the hole $\square$.
  - A frame can also be a *handler frame* $\#L^{P_{\text{op}}, L_{\text{env}}}\square$, which consists of a label L identifying the handler instance, a label P_{op} identifying the handler code, a label L_{env} identifying the closure environment, and a hole $\square$ where evaluation is in progress.
- A local environment E records the values of the arguments and local variables for the activation of a function.

To run a program, the initial configuration is constructed by loading the top-level function definitions into the code memory and invoking the main function.

The full set of reduction rules can be found in an appendix [21]. Figure 5 shows selected rules. Since the syntax is in A-normal form, the order of evaluation is already determined by the syntax, so there are no structural rules for locating the next redex.

The reduction rules can be categorized into two groups. Rules in the first group evaluate an expression locally. This group includes, for example, the NEWREF rule in Figure 5 for allocating a new tuple on the heap. The meta-level function $\hat{E}(v)$ is defined as $E(x)$ if $v = x$ and c if $v = c$. Rules in this group update the local environment E to reflect the new binding.

Rules in the second group involve control-flow transfer. These rules include APP, RET, HANDLE, LEAVE, RAISE, and RESUME. They either push frames to or pop frames from the evaluation context. They also switch to a different local environment, inserting new bindings to the new local environment to account for the arguments or the return value. We now take a closer look at the rules governing effect handling: HANDLE, LEAVE, RAISE, and RESUME.

$\boxed{\text{NEWREF}}$

$$\langle M \parallel H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{newref}\ (v_0, \cdots, v_{n-1})\ \texttt{in}\ t\rangle \longrightarrow$$
$$\left\langle M \parallel H[L \mapsto \left\langle \hat{E}(v_0), \cdots, \hat{E}(v_{n-1})\right\rangle] \parallel K \parallel E[x \mapsto L] \parallel t\right\rangle$$

where L is fresh

$\boxed{\text{HANDLE}}$

$$\left\langle M \parallel H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{handle}\ P_{\text{body}}\ \texttt{with}\ P_{\text{op}}\ \texttt{under}\ v_{\text{env}}\ \texttt{in}\ t\right\rangle \longrightarrow$$
$$\left\langle M \parallel H \parallel K \cdot (E, \texttt{let}\ x = \Box\ \texttt{in}\ t) \cdot \#L^{P_{\text{op}}, L_{\text{env}}}\Box \parallel [x_{\text{env}} \mapsto L_{\text{env}}, x_{\text{hdl}} \mapsto L] \parallel t'\right\rangle$$

where L is fresh, $M(P_{\text{body}}) = \lambda(x_{\text{env}}, x_{\text{hdl}}).\, t'$, and $\hat{E}(v_{\text{env}}) = L_{\text{env}}$

$\boxed{\text{LEAVE}}$

$$\left\langle M \parallel H \parallel K \cdot (E, \texttt{let}\ x = \Box\ \texttt{in}\ t) \cdot \#L^{P_{\text{op}}, L_{\text{env}}}\Box \parallel E' \parallel v\right\rangle \longrightarrow \left\langle M \parallel H \parallel K \parallel E[x \mapsto \hat{E}'(v)] \parallel t\right\rangle$$

$\boxed{\text{RAISE}}$

$$\left\langle M \parallel H \parallel K \cdot \left(\#L^{P_{\text{op}}, L_{\text{env}}}\Box\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise}\ v_1\ v_2\ \texttt{in}\ t\right\rangle \longrightarrow$$
$$\left\langle M \parallel H[L_k \mapsto \texttt{cont}\ \left(\#L^{P_{\text{op}}, L_{\text{env}}}\Box\right) \cdot K' \cdot (E, \texttt{let}\ x = \Box\ \texttt{in}\ t)] \parallel K \parallel \left[x_{\text{env}} \mapsto L_{\text{env}}, y \mapsto \hat{E}(v_2), k \mapsto L_k\right] \parallel t'\right\rangle$$

where L_k is fresh, $\hat{E}(v_1) = L$, and $M(P_{\text{op}}) = \lambda(x_{\text{env}}, y, k).\, t'$

$\boxed{\text{RESUME}}$

$$\langle M \parallel H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{resume}\ v_1\ v_2\ \texttt{in}\ t\rangle \longrightarrow$$
$$\left\langle M \parallel H[L_k \mapsto \texttt{ns}] \parallel K \cdot (E, \texttt{let}\ x = \Box\ \texttt{in}\ t) \cdot K' \parallel E'[x' \mapsto \hat{E}(v_2)] \parallel t'\right\rangle$$

where $\hat{E}(v_1) = L_k$ and $H(L_k) = \texttt{cont}\ K' \cdot (E', \texttt{let}\ x' = \Box\ \texttt{in}\ t')$

Figure 5. Selected reduction rules of LEXA.

The HANDLE rule pushes an activation frame consisting of the current local environment and the remaining term to the evaluation context. It also pushes a handler frame. The handler frame contains a freshly generated handler label L that identifies this newly installed handler instance. The rule creates a new local environment consisting of the two arguments that P_{body} receives: L_{env} is the closure environment of P_{body}, and L identifies the handler instance newly pushed onto the evaluation context. The body of the function P_{body} then becomes the term to be evaluated next. The generativity of the handler label L matches the semantics of previous languages supporting lexical handlers, such as Genus [35] and Effekt [7].

The LEAVE rule pops the handler frame from the evaluation context. It returns to the most recent activation frame and resumes the computation left there. The local environment is updated to reflect the return value.

The RAISE rule suspends the current computation and transfers control to a handler. The first operand of `raise` is interpreted into a handler label L identifying the handler instance to raise to. The second operand is the argument to the effect operation. A handler frame matching the label L is found in the evaluation context, and the evaluation context is unwound to that point. Further, a resumption is reified and stored in the heap. It is made up of the unwound frames and represents the suspended computation. A fresh label L_k identifies the resumption and can be used

location	ℓ	::=	P
word	v	::=	$\ell \mid i \mid \texttt{ns}$
operand	o	::=	$\mathbf{r} \mid v$
instruction	ι	::=	$\texttt{add}\ \mathbf{r}, o \mid \texttt{mkstk}\ \mathbf{r} \mid \texttt{salloc}\ i \mid \texttt{sfree}\ i \mid \texttt{malloc}\ \mathbf{r}_d, i \mid$
			$\texttt{mov}\ \mathbf{r}_d, o \mid \texttt{load}\ \mathbf{r}_d, [\mathbf{r}_s + i] \mid \texttt{store}\ [\mathbf{r}_d + i], o_s \mid$
			$\texttt{push}\ o \mid \texttt{pop}\ \mathbf{r} \mid \texttt{call}\ o \mid \texttt{jmp}\ o \mid \texttt{return} \mid \texttt{halt}$
instruction sequence	I	::=	$\cdot \mid \iota; I$
program	G	::=	$P_1 : I_1, \cdots, P_n : I_n$

Figure 6. Syntax of Salt.

location	ℓ	::=	$\cdots \mid L \mid \texttt{next}(\ell)$
code memory	M	::=	$\{P_1 \mapsto I_1, \cdots, P_n \mapsto I_n\}$
stack	s	::=	$\texttt{nil} \mid s :: v$
heap value	V	::=	$\langle v_1, \cdots, v_n \rangle \mid s$
heap	H	::=	$\{L_1 \mapsto V_1, \cdots, L_n \mapsto V_n\}$
register file	R	::=	$\{\mathbf{sp} \mapsto v_{\mathrm{sp}}, \mathbf{ip} \mapsto v_{\mathrm{ip}}, \mathbf{r1} \mapsto v_1, \cdots, \mathbf{rn} \mapsto v_n\}$
configuration	C	::=	$\langle M \parallel H \parallel R \rangle$

Figure 7. Syntax of the Salt abstract machine.

as a first-class value. We enforce dynamically that the resumption is resumed at most once, as we discuss soon. The handler code identified by P_{op} is the next redex. It accepts three arguments, so a new local environment is set up to run the handler code: L_{env} is the closure environment for P_{op}, $\hat{E}(v_2)$ is the argument to the effect operation, and L_k is the resumption.

The RESUME rule continues a suspended computation. The first operand of `resume` is interpreted into a label L_k identifying the resumption. The second operand is the argument to the resumption. A resumption $\texttt{cont}\ K' \cdot (E', \texttt{let}\ x' = \square\ \texttt{in}\ t')$ is found in the heap with the label L_k, and the frames K' are pushed onto the evaluation context. E', updated to bind x' to $\hat{E}(v_2)$, becomes the current local environment, and t' becomes the term to be evaluated next. To prevent the resumption from being resumed again, the heap is updated to map the label L_k to the nonsense value `ns`. Attempting to use L_k as a resumption will cause the evaluation to get stuck.

3.2 Target Language: Salt

Salt is an assembly-like language supporting heap-allocated stacks.

Syntax. Figure 6 presents the syntax of Salt. A *word* is either an address ℓ, an integer i, or nonsense `ns`. An *operand* is either a word v or a register name $\mathbf{r}$. *Instructions* are standard of an abstract assembly language. The only exception is the `mkstk` instruction, which is for allocating new stacks on the heap. A *program* G is a finite map from code addresses (ranged over by P) to instruction sequences. One of the code addresses is designated as the main function.

Operational semantics. We give an operational semantics to Salt as an abstract machine. The extra syntax needed for defining the machine's semantics is shown in Figure 7.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \texttt{mkstk}\ \mathbf{r} \quad L \text{ is fresh}}{\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H[L \mapsto \texttt{nil}] \parallel R[\mathbf{ip} \mapsto \texttt{next}(\ell), \mathbf{r} \mapsto L]\rangle}$$

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \texttt{salloc}\ n \quad R(\mathbf{sp}) = L^m \quad H(L) = \texttt{nil} :: v_1 :: \cdots :: v_m}{\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H[L \mapsto \texttt{nil} :: v_1 :: \cdots :: v_m :: \underbrace{\texttt{ns} :: \cdots :: \texttt{ns}}_{n}] \parallel R[\mathbf{ip} \mapsto \texttt{next}(\ell), \mathbf{sp} \mapsto L^{m+n}]\rangle}$$

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \texttt{mov}\ \mathbf{sp}, o \quad \hat{R}(o) = L^m \quad H(L) = \texttt{nil} :: v_1 :: \cdots :: v_m :: \cdots :: v_n}{\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H[L \mapsto \texttt{nil} :: v_1 :: \cdots :: v_m] \parallel R[\mathbf{ip} \mapsto \texttt{next}(\ell), \mathbf{sp} \mapsto L^m]\rangle}$$

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \texttt{call}\ o \quad R(\mathbf{sp}) = L^m \quad H(L) = \texttt{nil} :: v_1 :: \cdots :: v_m \quad \hat{R}(o) = \ell_{\text{dest}}}{\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H[L \mapsto \texttt{nil} :: v_1 :: \cdots :: v_m :: \texttt{next}(\ell)] \parallel R[\mathbf{ip} \mapsto \ell_{\text{dest}}, \mathbf{sp} \mapsto L^{m+1}]\rangle}$$

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \texttt{return} \quad R(\mathbf{sp}) = L^m \quad H(L) = \texttt{nil} :: v_1 :: \cdots :: v_{m-1} :: \ell_{\text{dest}}}{\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H[L \mapsto \texttt{nil} :: v_1 :: \cdots :: v_{m-1}] \parallel R[\mathbf{ip} \mapsto \ell_{\text{dest}}, \mathbf{sp} \mapsto L^{m-1}]\rangle}$$

Figure 8. Selected reduction rules of Salt.

A *stack* is a sequence of words ending with a special symbol nil that marks the *base* (as opposed to the *top*) of the stack. Stacks are heap-allocated; they are *heap values*, just like tuples.

An *address* can be a heap location L or a code-memory location P. Heap locations cannot appear in the program text. Locations can also be constructed through the next constructor. The notations ℓ^i and $\texttt{next}^i(\ell)$ are interchangeable, meaning $\texttt{next}(\cdots(\texttt{next}(\ell))\cdots)$ with i occurrences of next. For example, if L is the base of a stack in a heap H, meaning $H(L) = \texttt{nil} :: v_1 :: \cdots :: v_n$, then $L^j = \texttt{next}^j(L)$ is the stack location where v_j is stored ($1 \leq j \leq n$).

A *register file* R contains a finite number of registers. There are two distinguished registers: the stack-pointer register **sp** and the instruction-pointer register **ip**. We define a meta-level function $\hat{R}(o)$ such that $\hat{R}(\mathbf{r}) = R(\mathbf{r})$ and $\hat{R}(v) = v$.

A *code memory* M maps locations P to instruction sequences. If $M(P) = \iota_0; \cdots; \iota_{i-1}; I$, then the notations $\texttt{next}^i(P)$ and P^i mean the address of the instruction subsequence I. We define a meta-level partial function $\hat{M}(\ell)$ such that $\hat{M}(P^i) = \iota_i$ if $M(P) = \iota_0; \cdots; \iota_n$ ($0 \leq i \leq n$).

A machine *configuration* C consists of an immutable code memory, a mutable heap, and a mutable register file. Reduction takes the form $\langle M \parallel H \parallel R\rangle \longrightarrow \langle M \parallel H' \parallel R'\rangle$.

The full set of reduction rules can be found in an appendix [21]. Figure 8 shows selected rules that deal with stacks. To take a step, the instruction at the address stored in **ip** is executed. In each rule, the second premise specifies the instruction to execute. In case of a call, return, or jmp, **ip** is updated to reflect the nonlocal control transfer. In all other cases, **ip** is incremented.

The register **sp** always points to the top of a stack—an invariant respected by all the reduction rules. A mkstk instruction allocates a new stack on the heap. A salloc (resp. sfree) instruction grows (resp. shrinks) the stack top pointed to by **sp**. Newly allocated stack slots are initialized to ns. For an instruction mov $\mathbf{r}_d, o$, the word $\hat{R}(o)$ is copied into $\mathbf{r}_d$, and in case $\mathbf{r}_d$ is **sp** and $\hat{R}(o)$ is a stack location L^m, the stack L is cut to ensure that **sp** points to the stack top L^m. The opcodes call and return are like those in x86: they push or pop the return address onto or off the stack.

Notice that Salt does not have instructions specialized to effect handlers. Rather, effect handlers will be compiled to the low-level, general-purpose instructions of Salt. The low-level nature of

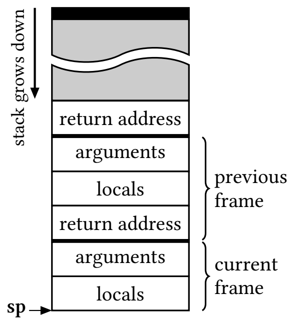


Figure 9. Layout of a single stack in SALT.

SALT distinguishes our approach from previous works [30, 32, 27] that define translations to formal models of high-level functional languages à la System F.

4 Translating LEXA to SALT

4.1 Overview

The translation from LEXA to SALT is defined with functions of the forms $[\![G]\!] = G$, $[\![\lambda\overline{x}.\, t]\!] = I$, $[\![t]\!]_\Gamma = I$, $[\![e]\!]_\Gamma = I$, and $[\![v]\!]^{\mathbf{r}}_\Gamma = \iota$, for translating LEXA programs, functions, terms, expressions, and values, respectively. For visual clarity, LEXA constructs are typeset in blue and SALT constructs in pink. The metavariable $\Gamma ::= \epsilon \mid \Gamma, x$ denotes a sequence of local variables in LEXA.

Figure 10 defines the translation for selected LEXA constructs. A full version can be found in an appendix [21]. $[\![\lambda\overline{x}.\, t]\!] = I$ translates a LEXA function into a SALT instruction sequence. It uses a simplified calling convention where all arguments are passed through registers and, upon entering a function, immediately pushed to the stack. Register **r1** is used to store the return value. Before leaving a function, the stack frame is deallocated by the callee. The last instruction `return` pops the return address from the previous frame and jumps to it.

The layout of a single stack is shown in Figure 9. The stack grows downwards. A black bar marks the base of a stack, and a thick black line marks the boundary between frames for visual clarity. Register **sp** always points to the top of the stack. The stack consists of a sequence of frames, with each frame starting with the arguments and local variables and ending with the return address; the most recent frame does not have a return address.

$[\![t]\!]_\Gamma = I$ translates a term t, with Γ providing bindings for the free variables in t. The result of evaluating the term t is left in **r1**. $[\![e]\!]_\Gamma = I$ translates an expression e, putting the result in **r1**. Most cases of this translation are ordinary, except for `handle`, `raise`, and `resume` expressions. $[\![v]\!]^{\mathbf{r}}_\Gamma = \iota$ translates a value v to an instruction ι that stores the word in the provided register **r**.

4.2 Translating `handle`, `raise`, and `resume` with Trampolines

We now illustrate the translation of `handle`, `raise`, and `resume` expressions. Most of the work happens in the built-in trampoline functions P_{handle}, P_{raise}, and P_{resume}, which are provided in the right column of Figure 10 as a reference.

To understand what the trampoline functions do, we illustrate concrete executions of the scheduler program (Figure 1) as control goes in and out of the trampoline functions. Figure 11 contains three subfigures, each illustrating the execution of a call to `handle`, `raise`, and `resume`, respectively. Each subfigure contains five state snapshots at different points during the execution of a trampoline: (1) before the trampoline is called, (2) at the beginning of the trampoline, (3) inside the trampoline, (4) before control is about to leave the trampoline, and (5) after control leaves the trampoline.

$$\boxed{[\![G]\!] = G}$$

$$[\![\texttt{letrec}\ P_1 = \lambda\overline{x_1}.\,t_1, \cdots, P_n = \lambda\overline{x_n}.\,t_n]\!] =$$
$$\quad P_1 : [\![\lambda\overline{x_1}.\,t_1]\!], \cdots, P_n : [\![\lambda\overline{x_n}.\,t_n]\!],$$
$$\quad P_{\text{handle}} : I_{\text{handle}}, P_{\text{handle\_special}} : I_{\text{handle\_special}},$$
$$\quad P_{\text{raise}} : I_{\text{raise}}, P_{\text{resume}} : I_{\text{resume}}$$

$$\boxed{[\![\lambda\overline{x}.\,t]\!] = I}$$

$$[\![\lambda(x_1, \cdots, x_n).\,t]\!] =$$
$$\quad \texttt{push}\ \mathbf{rn};\ \cdots;\ \texttt{push}\ \mathbf{r1};\ [\![t]\!]_{x_n,\cdots,x_1};\ \texttt{sfree}\ k;\ \texttt{return}$$
$$\quad \text{where } k = n + |t|$$

$$\boxed{[\![t]\!]_\Gamma = I}$$

$$[\![\texttt{let}\ x = e\ \texttt{in}\ t]\!]_\Gamma = [\![e]\!]_\Gamma;\ \texttt{push}\ \mathbf{r1};\ [\![t]\!]_{\Gamma,x}$$
$$[\![v\ \texttt{end}]\!]_\Gamma = [\![v]\!]^{\mathbf{r1}}_\Gamma$$

$$\boxed{[\![e]\!]_\Gamma = I}$$

$$[\![v]\!]_\Gamma = [\![v]\!]^{\mathbf{r1}}_\Gamma$$
$$[\![v_0(v_1, \cdots, v_n)]\!]_\Gamma = [\![v_n]\!]^{\mathbf{rn}}_\Gamma;\ \cdots;\ [\![v_0]\!]^{\mathbf{r0}}_\Gamma;\ \texttt{call}\ \mathbf{r0}$$
$$[\![\texttt{handle}\ P_{\text{body}}\ \texttt{with}\ A\ P_{\text{op}}\ \texttt{under}\ v_{\text{env}}]\!]_\Gamma =$$
$$\quad [\![A]\!]^{\mathbf{r4}};\ [\![v_{\text{env}}]\!]^{\mathbf{r3}}_\Gamma;\ \texttt{mov}\ \mathbf{r2}, P_{\text{op}};\ \texttt{mov}\ \mathbf{r1}, P_{\text{body}};$$
$$\quad \texttt{call}\ P_{\text{handle}}$$
$$[\![\texttt{raise}\ A\ v_1\ v_2]\!]_\Gamma = [\![v_2]\!]^{\mathbf{r2}}_\Gamma;\ [\![v_1]\!]^{\mathbf{r1}}_\Gamma;\ \texttt{call}\ P_{\text{raise}}$$
$$[\![\texttt{resume}\ v_1\ v_2]\!]_\Gamma = [\![v_2]\!]^{\mathbf{r2}}_\Gamma;\ [\![v_1]\!]^{\mathbf{r1}}_\Gamma;\ \texttt{call}\ P_{\text{resume}}$$

$$\boxed{[\![v]\!]^{\mathbf{r}}_\Gamma = \iota}$$

$$[\![x_i]\!]^{\mathbf{r}}_{x_n,\cdots,x_0} = \texttt{load}\ \mathbf{r}, [\mathbf{sp} - i]$$
$$[\![i]\!]^{\mathbf{r}}_\Gamma = \texttt{mov}\ \mathbf{r}, i$$
$$[\![P]\!]^{\mathbf{r}}_\Gamma = \texttt{mov}\ \mathbf{r}, P$$

$$\boxed{[\![A]\!]^{\mathbf{r}} = \iota}$$

$$[\![\texttt{general}]\!]^{\mathbf{r}} = \texttt{mov}\ \mathbf{r},\ 0$$
$$[\![\texttt{tail}]\!]^{\mathbf{r}} = \texttt{mov}\ \mathbf{r},\ 1$$
$$[\![\texttt{abort}]\!]^{\mathbf{r}} = \texttt{mov}\ \mathbf{r},\ 2$$

```
P_handle :                      # r1: P_body, r2: P_op, r3: L_env, r4 : A
0   mov r5, sp;                 # save old stack pointer
1   mkstk sp;                   # create new stack and switch to it
2-5 push r2; push r3; push r4; push r5;
                                # create header frame
6   mov r6, r1;                 # r6: P_body
7   mov r2, sp;                 # r2: exchanger slot
8   mov r1, r3;                 # r1: L_env
9   call r6;                    # call P_body with args in r1, r2
10  pop r2;                     # r2: top of parent stack
11  sfree 3;                    # deallocate remaining header frame
12  mov sp, r2;                 # switch stacks
13  return

P_raise :                       # r1: exchanger slot, r2: op arg
0   load r4, [r1];              # r4: top of handler stack
1   store [r1], sp;
                                # exchanger: top of resumption stack
2   mov sp, r4;                 # switch stacks
3   load r5, [r1 − 3];          # r5: P_op
4-5 malloc r3, 1; store [r3], r1;   # r3: L_k
6   load r1, [r1 − 2];          # r1: L_env
7   jmp r5;                     # call P_op with args in r1, r2, r3

P_resume :                      # r1: L_k, r2: resumption arg
0   load r3, [r1];              # r3: exchanger slot
1   store [r1], ns;             # invalidate resumption object
2   load r4, [r3];              # r4: top of resumption stack
3   store [r3], sp;             # exchanger: top of handler stack
4   mov sp, r4;                 # switch stacks
5   mov r1, r2;                 # r1: resumption arg
6   return
```

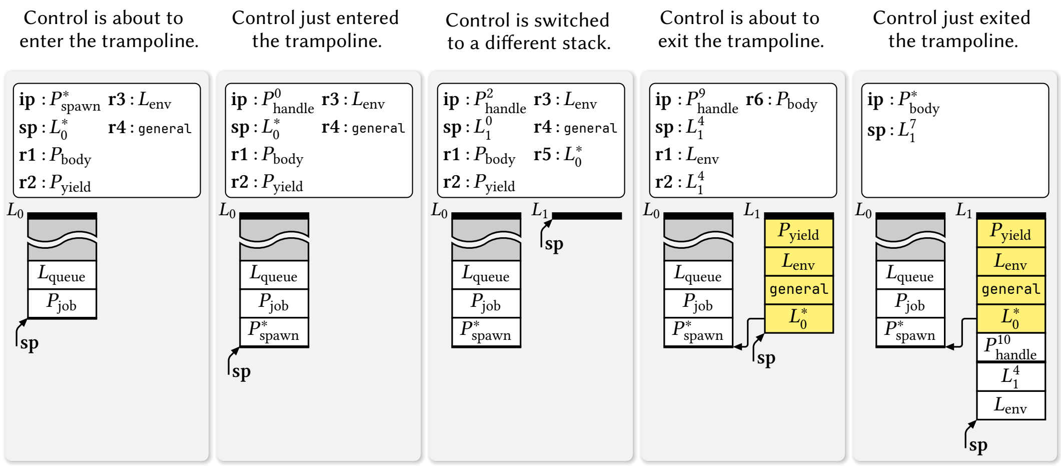
Figure 10. Left: Translation of selected constructs. Right: Built-in trampolines used by the translated code.

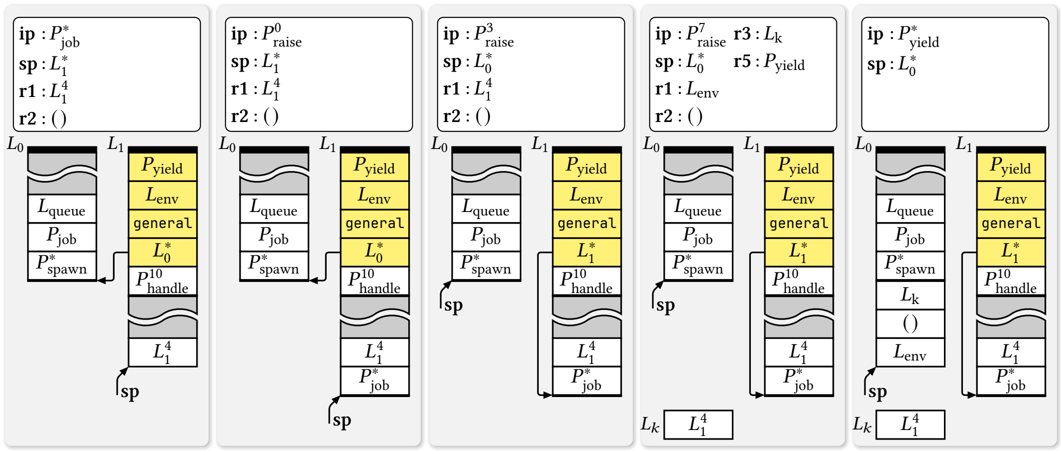
Each state snapshot consists of the register file and the heap-allocated stacks. Registers and stacks unrelated to the discussion are omitted. Stack slots that belong to *header frames* (explained later) are highlighted in yellow. The use of $*$ as a location offset (e.g., as in P^*_{spawn}) means that the exact offset is not relevant to the discussion.

As the readers go through the discussion, they should consult Figures 1 and 10. It helps to pay attention to the **ip** register, which points to the next instruction to execute.

Translating `handle` with P_{handle}. The state snapshots in Figure 11(a) shows the state transitions as control enters the `handle` expression and a handler instance is installed (line 18, Figure 1).

(1) In the first snapshot, control is currently inside P_{spawn} and is about to enter the trampoline P_{handle}, and the registers store the arguments to P_{handle}.


Control is about to enter the trampoline.
Control just entered the trampoline.
Control is switched to a different stack.
Control is about to exit the trampoline.
Control just exited the trampoline.
ip : P^*_{spawn} r3 : L_{env} sp : L^*_0 r4 : general r1 : P_{body} r2 : P_{yield}
ip : P^0_{handle} r3 : L_{env} sp : L^*_0 r4 : general r1 : P_{body} r2 : P_{yield}
ip : P^2_{handle} r3 : L_{env} sp : L^0_1 r4 : general r1 : P_{body} r5 : L^*_0 r2 : P_{yield}
ip : P^9_{handle} r6 : P_{body} sp : L^4_1 r1 : L_{env} r2 : L^4_1
ip : P^*_{body} sp : L^7_1
L_0 L_1 L_{queue} P_{job} P^*_{spawn} P_{yield} L_{env} general L^*_0 P^{10}_{handle} L^4_1 sp
(a) Invoking the P_{handle} trampoline as a result of executing a handle expression (cf. line 18, Figure 1).


ip : P^*_{job} sp : L^*_1 r1 : L^4_1 r2 : ()
ip : P^0_{raise} sp : L^*_1 r1 : L^4_1 r2 : ()
ip : P^3_{raise} sp : L^*_0 r1 : L^4_1 r2 : ()
ip : P^7_{raise} r3 : L_k sp : L^*_0 r5 : P_{yield} r1 : L_{env} r2 : ()
ip : P^*_{yield} sp : L^*_0
L_0 L_1 L_k L_{queue} P_{job} P^*_{spawn} P_{yield} L_{env} general L^*_0 L^*_1 P^{10}_{handle} L^4_1 P^*_{job} () sp
(b) Invoking the P_{raise} trampoline as a result of executing a raise expression (cf. line 31, Figure 1).

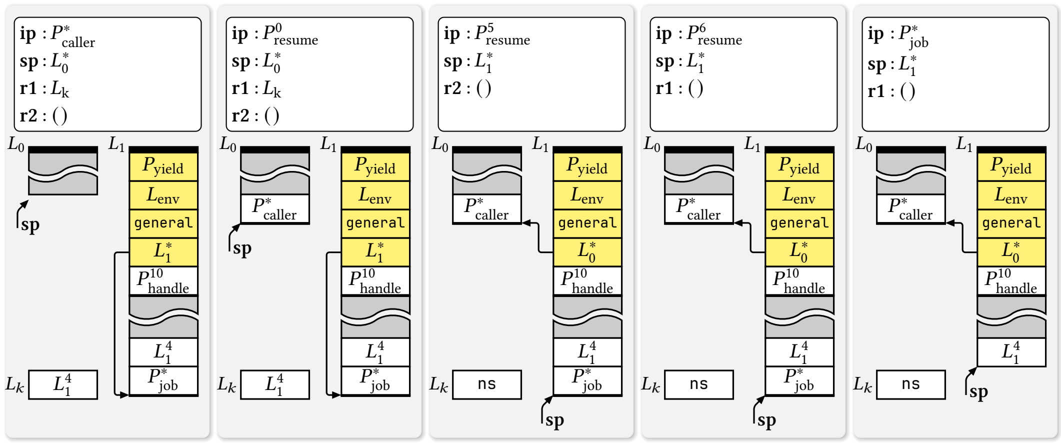

ip : P^*_{caller} sp : L^*_0 r1 : L_k r2 : ()
ip : P^0_{resume} sp : L^*_0 r1 : L_k r2 : ()
ip : P^5_{resume} sp : L^*_1 r2 : ()
ip : P^6_{resume} sp : L^*_1 r1 : ()
ip : P^*_{job} sp : L^*_1 r1 : ()
L_0 L_1 L_k P^*_{caller} P_{yield} L_{env} general L^*_1 L^*_0 P^{10}_{handle} L^4_1 P^*_{job} ns sp
(c) Invoking the P_{resume} trampoline as a result of executing a resume expression (cf. line 12, Figure 1).

Figure 11. State snapshots during the execution of the program in Figure 1, illustrating the SALT trampolines responsible for stack switching.

(2) Upon the instruction `call` P_{handle}, the return address is pushed to the stack L_0, as shown in the second snapshot.
(3) Next, a new stack is allocated, as shown in the third snapshot.
(4) Four values are pushed onto the new stack: the address of the handler operation P_{yield} (P_{fork} is omitted in the presentation), the address of the closure environment L_{env}, the handler annotation `general` (Section 5 discusses handler annotations), and the old value of **sp**. We call this region of the stack the *header frame*. In particular, we call the fourth slot of the header frame, which currently stores the old value of **sp**, the *exchanger*. As we will see later, the exchanger always points to the top of some stack: either the top of the parent stack or the top of the resumption stack. It facilitates the exchange of stack pointers during a `raise` or `resume`.
Now control is about to be transferred from P_{handle} to P_{body}, the code being handled (line 19, Figure 1). P_{body} expects two arguments, which have been moved into the registers **r1** and **r2**: the closure environment L_{env} and the location L_1^4 of the newly allocated header frame. Here, the location of the header frame is used in P_{body} to identify the handler instance.
(5) As control enters P_{body} in the final snapshot, the return address and the function arguments are pushed onto the stack.

Translating `raise` with P_{raise}. The state snapshots of Figure 11(b) show the state transitions as an effect is raised (line 31, Figure 1), a resumption is captured, and control is transferred to a handler.
(1) In the first snapshot, control is currently inside P_{job} and is about to enter the trampoline P_{raise}, and the registers store the arguments to P_{raise}: **r1** points to the header frame of the handler, and **r2** stores the argument to the effect operation.
(2) Upon the instruction `call` P_{raise}, the return address is pushed to the stack L_1, as shown in the second snapshot.
(3) Next, the contents of the exchanger and **sp** are swapped, as shown in the third snapshot.
(4) In the fourth snapshot, control is about to leave P_{raise} for the handler code P_{yield}. A single-element tuple is allocated to store the address of the header frame. The address of this tuple, L_k, serves as the identity of the resumption. L_k is stored in **r3** as the last argument to P_{yield}.
(5) Finally, control enters P_{yield} in the fifth snapshot. Its arguments are pushed onto the stack L_0.
Notice that, importantly, no run-time search or stack walking is needed to identify the handler or capture the resumption.

Translating `resume` with P_{resume}. The state snapshots of Figure 11(c) show the state transitions as a previously captured resumption is resumed (line 12, Figure 1).
(1) In the first snapshot, control is currently inside the caller of `resume` and is about to enter the trampoline P_{resume}. The registers store the arguments to P_{resume}: **r1** points to the resumption object, and **r2** stores the argument to the resumption.
(2) Upon the instruction `call` P_{resume}, the return address is pushed to the stack L_0, as shown in the second snapshot.
(3) Next, the pointer to the exchanger in the header frame is loaded from the resumption object at L_k. The contents of the exchanger and **sp** are then swapped. The resumption object is then updated to prevent it from being resumed again; this is shown in the third snapshot.
(4) In the fourth snapshot, control is about leave P_{resume} for the resumption (i.e., the previously suspended computation in P_{job}). The argument to the resumption is moved to **r1**.
(5) Finally, a `return` instruction is executed, which pops the return address off the stack L_1 and transfers control to the suspended computation in P_{job}, as shown in the fifth snapshot.

$$
\begin{array}{llll}
\text{handler annotation} & A & ::= & \texttt{tail} \mid \texttt{abort} \mid \texttt{general} \\
\text{expression} & e & ::= & \cdots \mid \texttt{handle}\ P_{\text{body}}\ \texttt{with}\ A\ P_{\text{op}}\ \texttt{under}\ v_{\text{env}} \mid \texttt{raise}\ A\ v_1\ v_2 \\
\text{frame} & F & ::= & \cdots \mid \#L_{A}^{P_{\text{op}},L_{\text{env}}}\square
\end{array}
$$

$$
\begin{array}{l}
\left\langle M \parallel H \parallel K \cdot \left(\#L_{\texttt{tail}}^{P_{\text{op}},L_{\text{env}}}\square\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise tail}\ v_1\ v_2\ \texttt{in}\ t\right\rangle \longrightarrow \\
\left\langle M \parallel H \parallel K \cdot \left(\#L_{\texttt{tail}}^{P_{\text{op}},L_{\text{env}}}\square\right) \cdot K' \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \parallel \left[x_{\text{env}} \mapsto L_{\text{env}}, y \mapsto \hat{E}(v_2)\right] \parallel t'\right\rangle \\
\text{where } \hat{E}(v_1) = L \text{ and } M(P_{\text{op}}) = \lambda(x_{\text{env}}, y).t'
\end{array}
$$

$$
\begin{array}{l}
\left\langle M \parallel H \parallel K \cdot \left(\#L_{\texttt{abort}}^{P_{\text{op}},L_{\text{env}}}\square\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise abort}\ v_1\ v_2\ \texttt{in}\ t\right\rangle \longrightarrow \\
\left\langle M \parallel H \parallel K \parallel \left[x_{\text{env}} \mapsto L_{\text{env}}, y \mapsto \hat{E}(v_2)\right] \parallel t'\right\rangle \\
\text{where } \hat{E}(v_1) = L \text{ and } M(P_{\text{op}}) = \lambda(x_{\text{env}}, y).t'
\end{array}
$$

$[\![\texttt{handle}\ P_{\text{body}}\ \texttt{with}\ A\ P_{\text{op}}\ \texttt{under}\ v_{\text{env}}]\!]_\Gamma =$
$\quad [\![A]\!]^{\mathbf{r4}}\ ;\ [\![v_{\text{env}}]\!]_\Gamma^{\mathbf{r3}}\ ;$
$\quad \texttt{mov}\ \mathbf{r2}, P_{\text{op}};$
$\quad \texttt{mov}\ \mathbf{r1}, P_{\text{body}};$
$\quad \texttt{call}\ P_{\text{handle\_special}}$
$\quad$ where $A = \texttt{tail}$ or $A = \texttt{abort}$

$[\![\texttt{raise tail}\ v_1\ v_2]\!]_\Gamma =$
$\quad [\![v_2]\!]_\Gamma^{\mathbf{r2}}\ ;\ [\![v_1]\!]_\Gamma^{\mathbf{r1}}\ ;$
$\quad \texttt{load}\ \mathbf{r3}, [\mathbf{r1} - 3];$
$\quad \texttt{load}\ \mathbf{r1}, [\mathbf{r1} - 2];$
$\quad \texttt{call}\ \mathbf{r3}$

$[\![\texttt{raise abort}\ v_1\ v_2]\!]_\Gamma =$
$\quad [\![v_2]\!]_\Gamma^{\mathbf{r2}}\ ;\ [\![v_1]\!]_\Gamma^{\mathbf{r1}}\ ;$
$\quad \texttt{load}\ \mathbf{r3}, [\mathbf{r1} - 3];$
$\quad \texttt{mov}\ \mathbf{sp}, \mathbf{r1};$
$\quad \texttt{load}\ \mathbf{r1}, [\mathbf{r1} - 2];$
$\quad \texttt{sfree}\ 4;\ \texttt{jmp}\ \mathbf{r3}$

$P_{\text{handle\_special}}$:
```
    push r2; push r3; push r4; push ns;          # r1: P_body, r2: P_op, r3: L_env, r4 : A
                                                 # create header frame on same stack
    mov r6, r1; mov r2, sp; mov r1, r3; call r6; # call P_body with args in r1 and r2
    sfree 4; return                              # deallocate header frame and return
```

Figure 12. Extending Lexa and the Lexa-to-Salt translation to address tail-resumptive and abortive handlers. The HANDLE and RAISE rules are omitted as they require minimal changes to the rules in Figure 5.

5 Optimizing Tail-Resumptive and Abortive Handlers

Entering a `handle` expression in general requires heap-allocating a new stack. However, this allocation is not necessary when the handler is either tail-resumptive or abortive. A *tail-resumptive* handler resumes the captured resumption in a tail position, and an *abortive* handler does not resume the resumption at all. In practice, many handlers are tail-resumptive (e.g., the client code of a coroutine-style iterator) or abortive (e.g., exception handlers). So it is worthwhile to optimize the compilation of these handlers to make them as efficient as possible. This section formalizes how our compilation strategy avoids allocating new stacks for tail-resumptive and abortive handlers and how tail-resumptive handlers can be invoked in place.

Syntax. To capture the essence of how these special handlers are translated, we update the syntax of Lexa, as Figure 12 shows. `handle` expressions, as well as handler frames, are extended with annotations A that indicate whether the handler is tail-resumptive, abortive, or otherwise. To simplify the formalism, `raise` expressions are also extended with annotations A, and the operational semantics requires that a handler should only handle effects raised by a `raise` expression with the same annotation. Our implementation of the Lexa compiler is more flexible, though; it does not require `raise` expressions to be annotated. Rather, the `raise` implementation looks up the annotation in the handler's header frame and dispatches accordingly.

Operational semantics. The HANDLE rule in Figure 5 still works for all three kinds of handlers; it only needs to be updated to include a metavariable A. The RAISE rule in Figure 5 is updated to work only for `raise general` expressions. Two additional rules are added for `raise tail` and `raise abort` expressions, as Figure 12 shows.

A `raise tail` expression is reduced *in place* as if it is a regular function call. A `raise abort` expression is reduced by aborting the surrounding computation delimited by the handler frame. Notice that in both cases, the handler implementation $M(P_{\text{op}})$ is no longer parameterized by a continuation k. Neither case reifies the resumption as a heap-allocated object, which justifies the optimization of not allocating new stacks in the translation.

Importantly, even though tail-resumptive handlers are raised in place, they can readily raise their own effects without any extra bookkeeping, thanks to the lexical scoping of handlers. In languages where handlers are dynamically scoped, in-place invocation of tail-resumptive handlers is tricky to get right: extra care must be taken to ensure that the in-place invocation of a handler does not bring in unintended handlers in the dynamic context. For example, in Xie and Leijen [32], the tail-resume optimization requires a special `under` construct, which filters out any handlers that should be skipped when the tail-resumptive handler raises its own effects. In contrast, with lexical scoping, a tail-resumptive handler has already captured the right handlers in its lexical closure, so it can be invoked directly without extra machinery! This simplicity speaks to the well-behaved nature of the lexical-scoping semantics.

Translation. Figure 12 shows that the translation of `handle` expressions is specialized for tail-resumptive and abortive handlers: it allocates the header frame on the same stack as the parent. Instead of P_{handle}, we use the trampoline $P_{\text{handle\_special}}$ to translate a `handle` expression when the handler is declared `tail` or `abort`. Notice that the implementation of $P_{\text{handle\_special}}$ does not use any `mkstk` instructions. Also notice that $P_{\text{handle\_special}}$ pushes an `ns` value onto the stack in place of the exchanger, as the exchanger is needed only when the resumption has to be reified and stacks switched.

Figure 12 also shows how `raise tail` and `raise abort` expressions are translated. The translation of the former looks like a regular function call. The translation of the latter cuts the stack to the abortive handler's header frame and then jumps to the handler's code.

6 Correctness of the Lexa-to-Salt Translation

To show that the translation is semantics-preserving, we give a simulation proof that relates the execution of a program in Lexa to the execution of the translated program in Salt. Our theoretical framework strictly follows Leroy [18].

First, we define two predicates, initial and final, for both languages. The initial(G, C) predicate relates a program G with its initial configuration C, and the final(C, i) predicate relates a terminal configuration C with the result i of the program.

Definition 1 *(Initial and final configurations).* The predicates $\text{initial}_{\text{Lexa}}(G, C)$, $\text{initial}_{\text{Salt}}(G, C)$, $\text{final}_{\text{Lexa}}(C, i)$, and $\text{final}_{\text{Salt}}(C, i)$ are defined as follows:

- $\text{initial}_{\text{Lexa}}\left(\texttt{letrec}\ \overline{P_i = \lambda \overline{x_i}.\, t_i},\ \left\langle \left\{ \overline{P_i \mapsto \lambda \overline{x_i}.\, t_i}, P_{\text{init}} \mapsto t_{\text{init}} \right\} \parallel \{\} \parallel \square \parallel [\,] \parallel t_{\text{init}} \right\rangle\right)$,
  where t_{init} is defined as $\texttt{let}\ x = P_{\text{main}}\ ()\ \texttt{in let}\ \_ = \texttt{exit}\ x\ \texttt{in ns end}$
- $\text{initial}_{\text{Salt}}\left(\overline{P_i : I_i},\ \left\langle \left\{ \overline{P_i \mapsto I_i},\ P_{\text{init}} \mapsto I_{\text{init}} \right\} \parallel \{L_{\text{init}} \mapsto \texttt{nil}\} \parallel \{\mathbf{sp} \mapsto L_{\text{init}}, \mathbf{ip} \mapsto P_{\text{init}}\} \right\rangle\right)$,
  where I_{init} is defined as $[\![\lambda().\, t_{\text{init}}]\!]$

- $\text{final}_{\text{LEXA}}\ (\langle M \parallel H \parallel K \parallel [x_{\text{result}} \mapsto i] \parallel \texttt{halt}\rangle, i)$
- $\text{final}_{\text{SALT}}\ \big(\big\langle M \parallel H[L_{\text{sp}} \mapsto\ s :: i] \parallel R[\mathbf{ip} \mapsto \ell, \mathbf{sp} \mapsto L_{\text{sp}}]\big\rangle, i\big)$ where $\hat{P}(\ell) = \texttt{halt}$

The initial configuration of Lexa is set up as follows. The code memory is initialized to the top-level functions, along with an additional P_{init} that calls the main function and exits. The heap, the evaluation context, and the local environment are all empty.

The initial configuration of Salt is set up as follows. The code memory is initialized to the top-level instruction sequences declared in the program, along with an additional P_{init} that calls the main function and halts. The heap contains an initial empty stack with the base address L_{init}. In the register file, $\mathbf{sp}$ points to the initial stack L_{init}, and $\mathbf{ip}$ points to the initial instruction P_{init}.

In a final configuration of Lexa, the term is `halt`, and the value in the current local environment is considered as the result computed by the program.

In a final configuration of Salt, $\mathbf{ip}$ points to a `halt` instruction, and the value at the top of the active stack is considered as the result computed by the program.

Next, we define the *observable behaviors* of programs of the two languages. Starting from an initial configuration, if after a finite sequence of reductions, the configuration becomes final with result i, the program is said to have the observational behavior converge(i). If the program gets stuck, the program is said to have the observational behavior stuck. If the program neither converges nor gets stuck after a finite sequence of reductions, the program is said to have the observational behavior diverge.

Definition 2 (Observable behavior). Given an initial(G, C) relation, a final(C, i) relation, and a reduction relation $\longrightarrow$ for a language, the *observable behavior* B of a program G in the language is defined with the notation $G \Downarrow B$ as follows:

$$B ::= \text{converge}(i) \mid \text{stuck} \mid \text{diverge}$$

$$\boxed{G \Downarrow B}$$

$$\frac{\text{initial}(G, C) \qquad C \longrightarrow^* C' \qquad \text{final}(C', i)}{G \Downarrow \text{converge}(i)} \qquad \frac{\text{initial}(G, C) \qquad C \longrightarrow^* C' \qquad C' \not\longrightarrow \qquad \forall i, \neg\text{final}(C', i)}{G \Downarrow \text{stuck}} \qquad \frac{\text{initial}(G, C) \qquad C \longrightarrow^\infty}{G \Downarrow \text{diverge}}$$

With observable behaviors defined, we can define semantic preservation.

Definition 3 (Semantic preservation). A translation function $[\![\cdot]\!]$ is *semantics-preserving* if, whenever $G \Downarrow B$ and $B \neq \text{stuck}$, we have $[\![G]\!] \Downarrow B$.

To prove that our translation from Lexa to Salt preserves semantics, we use a simulation argument: we construct a relation $\sim$ that relates the configurations of Lexa to those of Salt, and show that the relation is preserved by the reduction relation of both languages. Before showing how the relation is constructed, which is presented in the rest of this section, we first give the necessary theoretical background. Below, we first formally define *simulation* and then state the theorem that reduces semantic preservation to simulation.

Definition 4 (Simulation). A target-language reduction relation $\longrightarrow$ *simulates* a source-language reduction relation $\longrightarrow$ with respect to a relation $\sim$ if, for all C_1, C_2, and C_1' such that $C_1 \sim C_2$ and $C_1 \longrightarrow C_1'$, there exists C_2' such that $C_2 \longrightarrow^+ C_2'$ and $C_1' \sim C_2'$.

Theorem 1. A function $[\![\cdot]\!]$ translating programs in a source language $\mathcal{L}_1$ to programs in a target language $\mathcal{L}_2$ is semantics-preserving, if given an $\mathcal{L}_1$ program G_1 and its translation $G_2 = [\![G_1]\!]$ in $\mathcal{L}_2$, the following conditions hold:

(1) If $\text{initial}_{\mathcal{L}_1}(G_1, C_1)$ and $\text{initial}_{\mathcal{L}_2}(G_2, C_2)$, then $C_1 \sim C_2$.
(2) If $C_1 \sim C_2$ and $\text{final}_{\mathcal{L}_1}(C_1, i)$, then $\text{final}_{\mathcal{L}_2}(C_2, i)$.
(3) $\longrightarrow$ simulates $\longrightarrow$ with respect to $\sim$.

The proof of Theorem 1 can be found in Leroy [18]. Given the theorem, all we need to do is to construct the simulation relation $\sim$ and show that it satisfies the three conditions.

We now construct the $C \sim C$ relation between Lexa and Salt configurations:

$$\frac{(E, t) \overset{\text{ins}}{\sim} I \qquad (K, E) \underset{\Xi}{\overset{\text{context}}{\sim}} (H_{\text{stacks}}, \ell_{\text{sp}}) \qquad H \underset{\Xi}{\overset{\text{data}}{\sim}} H \qquad M \overset{\text{code}}{\sim} M \qquad M(P) = \iota_0 :: \cdots \iota_j :: I}{\langle M \parallel H \parallel K \parallel E \parallel t \rangle \sim \langle M \parallel H_{\text{stacks}} \uplus H \parallel \{\mathbf{sp} \mapsto \ell_{\text{sp}}, \mathbf{ip} \mapsto P^{j+1}\} \rangle}$$

The relation is defined using several auxiliary relations that relate components of the configurations. We briefly describe them below; their definitions are omitted for brevity but can be found in an appendix [21]. Some auxiliary relations are indexed by a context Ξ that maps Lexa labels to Salt addresses. The mapping for most labels can be statically determined, except for the handler labels of tail-resumptive and abortive handlers. These labels correspond to addresses in the middle of a stack; the offset from the base of a stack is only known at run time.

The first premise $(E, t) \overset{\text{ins}}{\sim} I$ relates a Lexa term t to a Salt instruction sequence I. The relation also takes as input a Lexa local environment E: local variables in E are stack-allocated and need to be deallocated at the end of the instruction sequence I.

The second premise $(K, E) \underset{\Xi}{\overset{\text{context}}{\sim}} (H_{\text{stacks}}, \ell_{\text{sp}})$ connects the notions of the current point of control in the two languages. It relates a Lexa evaluation context K to a Salt heap H_{stacks}. An evaluation context in Lexa can correspond to multiple stacks in Salt, so H_{stacks} can contain multiple entries. The relation also relates a local environment E and the location pointed to by the stack pointer: E corresponds to the most recently pushed stack frame.

The third premise $H \underset{\Xi}{\overset{\text{data}}{\sim}} H$ relates a Lexa heap H to a Salt heap H. The Lexa heap contains tuples and captured resumptions. How Lexa tuples and Salt tuples are related is straightforward. As for resumptions, one resumption in Lexa corresponds to a single-element tuple and a collection of stacks in Salt. The correspondence between a Lexa resumption and a collection of Salt stacks is defined similarly to the relation $\underset{\Xi}{\overset{\text{context}}{\sim}}$ discussed earlier.

The fourth premise $M \overset{\text{code}}{\sim} M$ relates a code memory M in Lexa to a code memory M in Salt. The relation is straightforward.

The final premise states that the instruction sequence I, pointed to by the instruction pointer **ip**, must be a subsequence of a function in the code memory M.

Theorem 2. The translation from Lexa to Salt is semantics-preserving.

Theorem 2 is reduced to proving that the simulation relation $\sim$ satisfies the three conditions prescribed in Theorem 1. The proof is largely mechanical and can be found in the appendix.

7 The Lexa Compiler

The previous sections provide a formal account of the compilation process. In this section, we describe the implementation of our compiler for Lexa.

Unlike the formalization, which assumes programs have undergone closure conversion, our compiler supports a friendlier syntax for Lexa. The compiler applies closure conversion and hoisting to a Lexa program before further compiling it to C. The compiled C program is then fed to LLVM

```
1  queue_t* q = queueMake();
2  int state = 0;
3  int n_jobs = 1000;
4  void driver(){
5    resumption_t* k = queueDeq(q);
6    RESUME(k, 0);
7    driver();
8  }
9  void yield(resumption_t* k){
10   queueEnq(q, k);
11 }
12 void fork(void* g, resumption_t* k){
13   queueEnq(q, k);
14   spawn(g);
15 }
16 void spawn(void (*f)(header_t*)){
17   HANDLE(f, ({GENERAL, yield},
18              {GENERAL, fork}));
19 }
20 void scheduler(void (*f)(header_t*)){
21   spawn(f);
22   driver();
23 }
24 void job(header_t* P){
25   ···; RAISE(P, 0, ()); ···
26 }
27 void jobs(header_t* P, header_t* T){
28   for (int i = n_jobs; i > 0; i--){
29     RAISE(P, 1, (job));
30   }
31 }
32 int main(){
33   init_stack_pool();
34   scheduler(jobs);
35   destroy_stack_pool();
36 }
```

Figure 13. The scheduler example in C using StackTrek.

for optimization and code generation. The generated code is linked with Boehm–Demers–Weiser garbage collector [5]. The compilation from Lexa to C is syntax-directed and does not involve any optimization; we leave all standard optimizations to LLVM.

We implement a C library, called StackTrek, that provides low-level facilities for stack switching. Its design largely follows the formalization discussed in the previous sections. The StackTrek library consists of x86-64 assembly functions responsible for stack switching. It also provides a set of user-facing macros, which are useful in themselves to a C programmer who wants to write programs with lexical effect handlers.

7.1 Overview

StackTrek provides several macros for stack switching: `HANDLE`, `RAISE`, and `RESUME`. They correspond to the built-in functions P_{handle}, P_{raise}, and P_{resume} in the formal translation (Section 4). Figure 13 demonstrates how the macros are used in the compiled C code for the scheduler example in Figure 1. To improve readability, we use C's global variables and remove the `env` parameter from closures. We also remove the `Tick` and `Exn` handlers for simplicity. An actual program would need to handle the exceptional case of an empty queue at line 5.

`HANDLE` (line 17) takes two arguments: the code to be handled and the handler. The handler contains an implementation for each effect operation. It also contains an annotation for each operation implementation. Depending on the annotation, `HANDLE` may allocate a new stack and switch to it. It allocates a header frame and uses assembly to enter the code being handled. `HANDLE` contains several branches intended for handlers of different annotations. For any `handle` expression, the branch to take is always resolved at compile time, so `HANDLE` is essentially executed as a linear sequence of instructions.

```
__attribute__((naked, preserve_none))
int64_t save_switch_and_run_handler(intptr_t* env, int64_t arg, void* exec, void* func) {
  __asm__ (
    "movq 0(%%rdx), %%rax" // Load sp from the exchanger
    "movq %%rsp, 0(%%rdx)" // Save sp to the exchanger
    "movq %%rax, %%rsp" // Switch to the new stack
    "jmpq *%%rcx" // Call the handler; the first three arguments are already in the right registers
  );}
```

Figure 14. Stack-switching function for the `RAISE` macro.

`RAISE` (line 25 and 29) takes three arguments: the pointer to the header frame identifying the handler, the operation index, and the arguments. The operation index identifies which effect operation of the handler is being invoked. `RAISE` uses assembly to invoke the operation implementation and, depending on the annotation stored in the header frame, may involve a stack switch.

`RESUME` (line 6) takes two arguments: the resumption and the value that the resumption is to be resumed with. Our formal translation enforces at run time that a resumption can be resumed at most once. This restriction is similar to MultiCore OCaml [28], which supports single-shot resumptions but not multishot ones. As mentioned earlier, supporting multishot resumptions is a nongoal for us.

7.2 Limited Support for Multishot Resumptions

Although our formalization does not support multishot resumptions, our implementation does, with some restrictions. First, let's see why it is difficult for us to fully support multishot resumptions. Imagine a naive implementation where stacks are copied when a resumption is resumed. The original resumption remains in the heap, and the new copy is linked to the active stack. The problem is that within the copied stacks there can still be pointers to locations within the original resumption; it is impractical to search for and update all of them.

In the absence of multishot resumptions, the compiler can assume that all handler frames, regardless of whether they are on the active stack or in the heap, are identified by unique labels. Because the memory allocator assigns unique addresses when allocating handler header frames, the compiler can use these addresses to represent the corresponding handler labels in Lexa. This correspondence allows the compiler to translate `raise` into a constant-time operation leveraging the random-access memory.

However, in the presence of multishot resumptions, this assumption no longer holds. Two distinct header frames can correspond to the same label in Lexa but have different memory addresses. Nevertheless, our implementation still supports multishot resumptions under specific conditions. We require that a multishot resumption contain no more than one stack and that at most one copy of it be installed on the active stack at any time. This restriction removes the need to search the copied stack for pointers into the original stack. Despite this restriction, all the benchmark programs used by prior work [22] and by our evaluation (Section 8) can still be expressed idiomatically.

7.3 Stack-Switching Functions

At the core of each StackTrek macro is a call to a function containing inline assembly. We call these functions *stack-switching functions*. They are written as `naked` C functions so that the compiler does not generate a prologue or epilogue for them. As an example, the function in Figure 14 is responsible for the stack switch when an effect is raised with the `RAISE` macro. It is convenient that the assembly sequences are wrapped inside C functions, as the C compiler will respect calling-convention annotations (Section 7.4) and thus save and restore the register state during the call and return.

7.4 Calling Convention

In Salt, which is an abstract machine, registers are used only for shuttling values between the memory and the processor. But an actual machine uses registers to store the program state as well. During the activation of a function, registers can be used to store local variables and temporary values, and when control is transferred to another function, the values of the registers need to be preserved so that computation can continue when control returns. The System V calling convention for x86-64, used by most of the modern C compilers, prescribes that about half of the registers be callee-saved and the other half be caller-saved. This scheme works when control flows are all local. But in the presence of stack switching, a caller might be returned to from a different callee, which does not have the correct register state that the caller expects. A common approach is to use `setjmp` and `longjmp` to save and restore the register state as part of the stack-switching process.

The OCaml compiler has a different approach. It uses a calling convention where all registers are caller-saved. This design choice has implications for stack switching: it does not matter if control is returned to from a different callee, because the caller can restore the register state just by itself. As a result, stack switching is cheap, as no explicit saving and restoring needs to be done by the user program. We will refer to this calling convention as `preserve_none`.

However, the `preserve_none` calling convention is not without cost. It may lead to higher overhead than the System V calling convention. Consider a function A that uses a register **r** throughout its activation and calls a function B twice, where B does not use **r**. With `preserve_none`, **r** needs to be saved and restored before and after each call to B even though B does not use **r**, costing four instructions. In contrast, with the System V calling convention and with **r** being a callee-saved register, only two instructions are needed for saving and restoring **r** in the prologue and epilogue of A.

In our implementation of Lexa, we use a hybrid calling convention. The System V calling convention is used for most user functions, while `preserve_none` is used for the stack-switching functions and for functions that implement effect operations. The `preserve_none` attribute in function declarations (e.g., line 1 of Figure 14) serves as a hint to the LLVM compiler. The `preserve_none` calling convention simplifies the implementation of StackTrek, because the stack-switching functions are no longer responsible for saving and restoring callee-saved registers, which would normally be done using `setjmp` and `longjmp`. Using `preserve_none` also facilitates optimizations, as we discuss soon. Furthermore, as `preserve_none` is not used for most user functions, we obtain the usual performance profile associated with the System V calling convention.

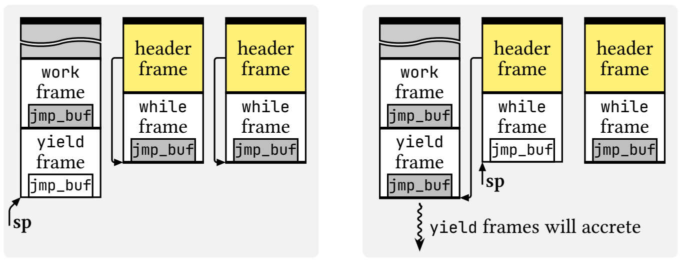
Lifting the overhead out of the hot path. When an effect operation is invoked in a hot path, the overhead of saving and restoring callee-saved registers can be significant. Consider line 29 in Figure 13, where the `fork` operation is invoked in a tight loop. If the System V calling convention is used, the callee-saved registers need to be saved inside the stack-switching function for `RAISE` every time `fork` is raised. This overhead will dominate the running time of the loop. In contrast, if the `preserve_none` calling convention is used for the stack-switching functions, since the compiler knows that calls to these functions can potentially clobber all registers, it includes every register in the use set of the caller. Consequently, the compiler will save and restore these registers in the prologue and epilogue of the caller, incurring the overhead only once.

Exposing tail-call optimization opportunities. The `preserve_none` calling convention enables tail-call optimizations that would otherwise be missed if `setjmp` and `longjmp` were used to implement stack switching.

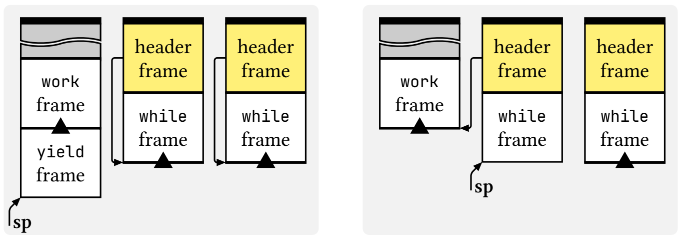
Consider the program in Figure 15(a). It implements round-robin scheduling between two tasks. When one task is running, the other task is parked in the heap. When the running task yields

```
1  val parked = ref None;
2  val work = fun() {
3    handle
4      while true
5        raise P.yield ()
6      done
7    with P : Process =
8    | yield _ k ⇒
9      val peer = !parked;
10     parked := Some k;
11     match peer with
12     | Some k’ ⇒
13       resume k’ ()
14     | None ⇒ ()
15   | fork g k ⇒ assert false
16 };
17 work();
18 work()
```

(a) Round-robin scheduling with effect handlers.



(b) Stack layout before/after resume, with setjmp and longjmp. A gray jmp_buf is populated.



(c) Stack layout before/after resume, with the Lexa compiler. A triangle marks the boundary from a SysV function to a preserve_none function.

Figure 15. For the program in (a), using setjmp and longjmp for stack switching can lead to stack overflow (b), while the preserve_none calling convention used by the Lexa compiler allows tail calls to be optimized (c).

(line 5), the parked task is resumed (line 13). Notice that the parked task is resumed in a tail position of the yield handler.[1] This scheduler is similar to the one in Sivaramakrishnan et al. [28, §3] in that a paused computation is resumed in a tail position of a handler.

Think about what happens if stack switching is implemented using setjmp and longjmp. When control goes from the main stack to a parked stack, callee-saved registers must be saved on the current stack in a struct of type jmp_buf. Later when control comes back, the registers must be restored from the struct. As a consequence, yield frames will accumulate on the stack, as Figure 15(b) illustrates. So the main stack grows with each yield and may eventually overflow (even if the jmp_buf structs are heap-allocated).

The problem is addressed by the preserve_none calling convention. The caller of a stack-switching function is required to save the register state before the call. Thus, the boundary between the caller and the callee demarcates a stack region that in itself records the information needed to resume the paused computation—these boundaries are indicated by the black triangles in Figure 15(c). Any code holding a stack pointer to a location marked by a black triangle can resume the computation at that location simply by executing a ret instruction, just as in our formalization.

The Lexa compiler translates resume k’ () at line 13 using the RESUME macro, which calls a stack-switching function at the end. Since the call is in a tail position, the compiler ensures that the yield frame will be popped off the main stack before the call.[2] Furthermore, since the stack-switching function itself does not have a frame, the stack size remains constant.

[1]This handler for yield is not tail-resumptive, however. Although k’ is resumed in a tail position of the handler, it is not the same resumption k of the current handler. So the tail-resume optimization does not apply.
[2]This tail-call optimization requires that the caller yield to also use the preserve_none calling convention, which is the case in our implementation as mentioned earlier.

7.5 Reducing Allocations

For efficiency, we aim to reduce heap allocations. Below, we discuss StackTrek's allocation strategies for different kinds of objects.

Stacklets. We use fixed-size stacklets for their simplicity, though StackTrek can be easily adapted to use a different allocation strategy [10]. Like prior work on implementing stacks and continuations [10, 28], we use a stack pool to recycle recently deallocated stacklets. The stack pool is initialized at program startup and is a contiguous block of memory divided into fixed-size stacklets. A global bitmap is used to keep track of the availability of the stacklets. When the runtime needs more stacklets than are available in the pool, it allocates a new block of memory on the spot. The stacklets are reclaimed using a combination of manual memory management and garbage collection. On the manual side, StackTrek releases stacklets back to the pool when it is certain that the stacklets are no longer needed. This can happen when the body of a `handle` expression completes. It can also happen when an abortive handler is invoked and control is transferred to an earlier stacklet. StackTrek also provides a special macro `RESUME_FINAL` for indicating that this is the last time a resumption is resumed. No copy of the stacklet will be made if it is installed with `RESUME_FINAL`, and the stacklet will be released to the pool after the resumption finishes. Stacklets that are not released manually are reclaimed by the garbage collector.

Header frames. In the formal translation from Lexa to Salt, a header frame contains a pointer to the environment of the handler, which is a heap-allocated object. StackTrek saves an allocation by allocating the environment directly as part of the header frame. This means that tail-resumptive and abortive handlers do not incur any heap allocation.

8 Evaluation

Setup of experiments. We evaluate our implementation of Lexa against other implementations of lexical effect handlers, on a benchmark suit maintained by the community [1]. We include three additional benchmarks: Resume Nontail 2, Scheduler, and Interruptible Iterator. The first tests the efficiency of capturing and resuming deep resumption stacks. The other two represent real-world uses of effect handlers and test the efficiency of effect propagation in the presence of deep stacks of handlers. The experiments were conducted on a workstation with a 3.5GHz CPU.

We compare Lexa with the two other languages that support lexical effect handlers: Effekt and Koka. Koka is most known for its support of dynamically scoped handlers, but it has recently incorporated support for *named handlers* [31], which are a form of lexical handlers. We also use as baselines two languages that support dynamically scoped handlers: Koka and OCaml. For Effekt and Koka, we use the latest versions available at the time of writing: Effekt 0.2.2, Koka 3.1.1, and OCaml 5.3.0. For OCaml, we use the multicont library [14] for multishot resumptions.

The Effekt compiler has multiple back ends. For most benchmarks, we use the MLton back end, as it produces the fastest code—MLton is a whole-program optimizing compiler for Standard ML [2]. However, the MLton back end cannot handle several benchmarks, due to a conflict between MLton's monomorphization requirement and the Effekt compiler's typed CPS translation: the benchmarks feature recursive, handler-polymorphic functions, which are CPS-translated by the Effekt compiler to stack-shape-polymorphic functions that cannot be monomorphized by MLton. For these benchmarks, we resort to the fastest back ends that can handle them: we use the Chez Scheme back end for Generator and Handler Sieve, and we use the Node.js back end for Scheduler and Interruptible Iterator. As Scheme and JavaScript run on virtual machines, running times were measured after a warm-up period.

Table 1. Benchmark running times measured for different systems. N/A indicates that the benchmark could not be implemented in the respective system. For those benchmarks that we could not implement with Effekt's MLton back end, we use the fastest back ends that can handle them (Chez Scheme or Node.js).

Benchmarks	Running time (ms)				
	Lexa	Effekt	Koka (named)	Koka (regular)	OCaml
Countdown	0	58	3721	2008	2468
Fibonacci Recursive	718	1659	1356	1346	1404
Product Early	146	279	1628	1636	150
Iterator	0	115	436	269	262
Nqueens	411	144	2070	1959	854
Generator	1261	1536 (Scheme)	9474	9337	1085
Tree Explore	189	292	293	285	172
Triples	267	35	1868	2336	424
Resume Nontail	170	113	1504	1434	265
Parsing Dollars	326	117	3042	3163	1888
Handler Sieve	570	3482 (Scheme)	2180	2216	2824
Resume Nontail 2	170	258	407620	406230	391
Scheduler	322	1892866 (JS)	N/A	2975	467
Interruptible Iterator	176	1225211 (JS)	1032	N/A	237190

The systems under comparison differ in many aspects, and the differences in performance can be attributed to many factors, not just the efficiency of the implementations of effect handlers. Most notably, Lexa has limited support for multishot resumptions, whereas the other systems support them without restrictions. In addition, Lexa, Effekt's MLton back end, and the OCaml compiler use garbage collectors, whereas Koka uses reference counting. Furthermore, Koka uses GCC as its back end, while Lexa uses LLVM. We advise the reader to take this potential for bias into account when interpreting the results.

Results of experiments. The results are presented in Table 1 and Figure 16. Table 1 shows the running times of the benchmarks for each of the systems under comparison. For the same benchmark, the same input size is used across the systems. Lexa is the fastest system on eight out of the 14 benchmarks. On the benchmarks where Lexa is not the fastest, it is the second fastest.

Figure 16 shows scaling plots for the benchmarks, with the input size on the x-axis and the running time on the y-axis. Lexa is the most scalable system on the same eight benchmarks.

Compared to Effekt, Lexa fares particularly well on three benchmarks: Handler Sieve, Scheduler, and Interruptible Iterator. These benchmarks involve recursion and thus lead to deep stacks of handlers at run time.

Also notable are the results for Countdown and Iterator, where Lexa is able to compile the program to a constant. Although most credit should be given to LLVM that carries out the heavy lifting of optimizations, that Lexa is able to generate code amenable to such optimizations is a testament to the quality of the generated code. In particular, because Lexa specially treats tail-resumptive handlers and allocates the header frame on the same stack, no assembly is involved in the Lexa-to-C compilation, allowing LLVM to perform optimizations without hindrance.

Lexa is not as efficient on benchmarks that use multishot resumptions (namely NQueens, Tree Explore, and Triples). On these benchmarks, Effekt is the fastest. Effekt represents resumptions as immutable closures and thus need not copy resumptions when they are used more than once.

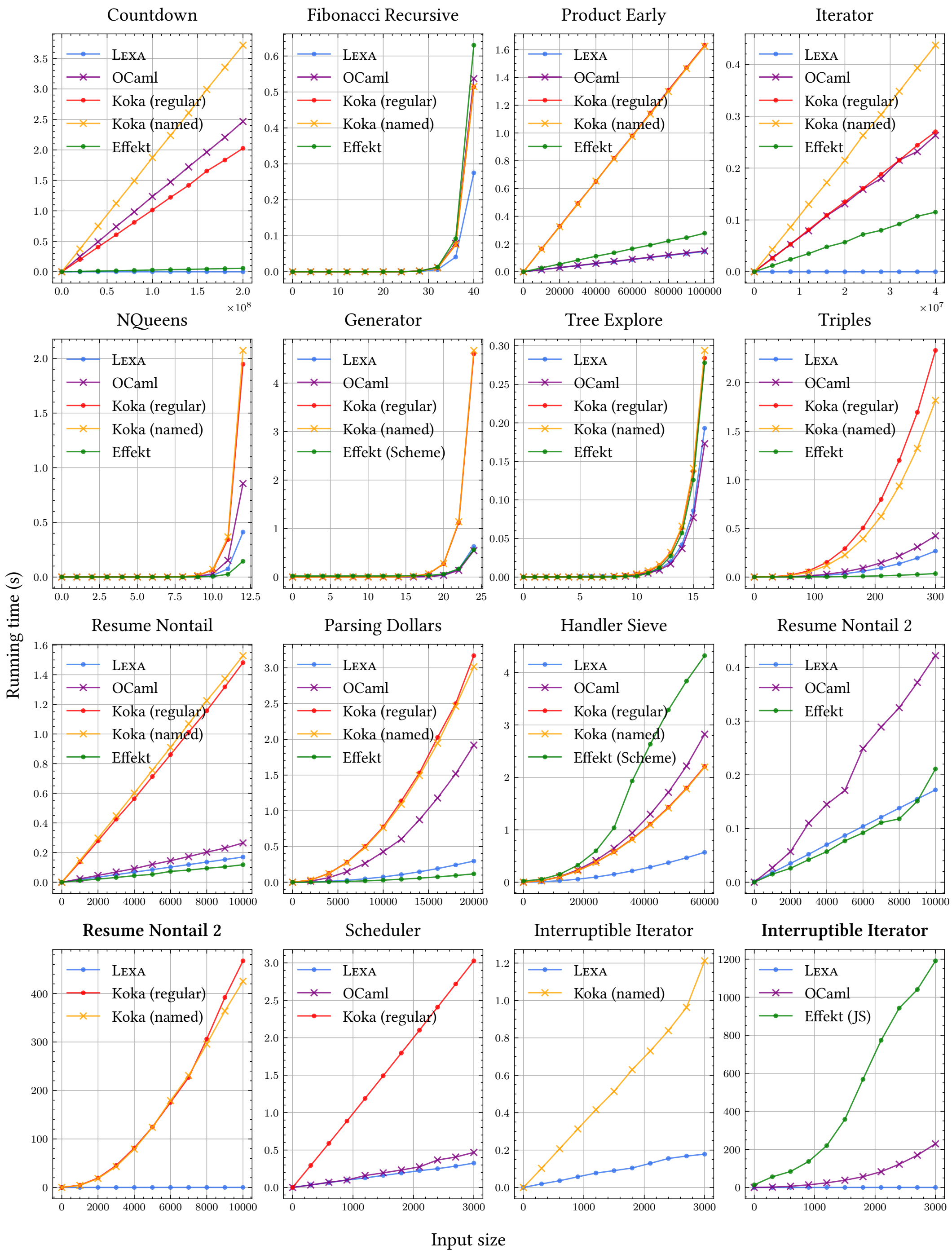


Figure 16. Scaling behaviors of different systems on 14 benchmarks. Effekt is omitted from the Scheduler chart; it has been shown in Figure 2. Resume Nontail 2 and Interruptible Iterator are each represented by two charts; the second chart (with the title in bold) reports on the systems that exhibit super-linear scaling behaviors on the benchmark, using Lexa as a reference point.

The Resume Nontail 2 benchmark differs from Resume Nontail in the original benchmark suite [1] in that an effectful function is made non-tail-recursive to force the stack to grow as the input size increases. This benchmark is interesting because it stresses the cost of capturing and resuming deep resumption stacks. Both Lexa and Effekt exhibit a linear scaling behavior on this benchmark. OCaml is piece-wise linear, which is likely a result of its stack management strategy. Koka (regular) and Koka (named) exhibit a super-linear scaling behavior, which suggests that capturing a resumption is not a constant-time operation in Koka. Koka builds up the resumption frame by frame, which takes longer as the stack grows.

The Scheduler benchmark, featuring `Tick` effects, is shown in Figure 1. We could not implement Scheduler with named handlers in Koka due to a compiler-internal error. Effekt does not scale on this benchmark, as discussed in Section 2. Interestingly, OCaml scales well: in OCaml, exception handling is implemented with a different mechanism than effect handling—by passing the code address and stack location of an exception handler to the raise site. Koka (regular) also scales linearly, but for a different reason: tail resumption. Because the `Tick` handler is tail-resumptive (line 53) and because Koka performs the tail-resume optimization, `Tick` can be invoked in place without reifying a resumption. However, if the tail-resume hint is removed from the `Tick` handler, Koka's performance degrades to super-linear scaling, because the cost of reifying a resumption increases with the input size.

The Interruptible Iterator benchmark is adapted from prior work [19, 36]. An implementation of this benchmark in Lexa is sketched in an appendix [21]. It uses bidirectional effects [36] to allow the client code of a coroutine iterator to concurrently update a list during iteration. The iterator code raises `Yield` effects to the client, which can then issue reverse-direction interrupts to the iterator: the client can replace the current element by raising a `Replace` effect, and it can remove the current element by raising a `Behead` effect. Like the Scheduler benchmark, Interruptible Iterator leads to $O(n)$ recursion depth and installs $O(n)$ handlers, where n is the length of the list being iterated. Unlike in Scheduler, it seems that the $O(n)$ installed handlers cannot be avoided simply by adjusting the lexical scope of the handler. Effekt does not scale well for this benchmark, requiring super-linearly increasing running time. Like Effekt, OCaml also scales super-linearly, as its effect-handling semantics requires walking the stack to find the `Yield` handler. We were able to implement Interruptible Iterator in Koka with named handlers, though not with regular handlers. We observe linear scaling with Koka's named handlers. But if the tail-resume hint is removed from the `Yield` handler, we observe super-linear scaling, which again suggests that reifying a resumption is not a constant-time operation in Koka.

9 Related Work

Zhang et al. [35] demonstrate that dynamically scoped exception handlers can lead to exceptions being caught by the wrong handler. To address this problem, they introduce *tunneled exceptions* and a type system enforcing that exceptions tunnel through program contexts oblivious to them. Tunneling is based on the principle of local reasoning; it is essentially a form of lexically scoped exception handlers. Zhang et al. [35] implement tunneled exceptions for the Genus programming language [33], compiling them to unchecked Java exceptions. Installing a handler generates a fresh identifier, which is passed down the call chain as a capability to raise exceptions to that handler. Upon an exception, the call stack is walked to find a handler with the matching identifier.

Zhang and Myers [34] suggest that the loss of local reasoning with dynamically scoped effect handlers can be analogized to a loss of *parametricity* [29]. They prove, for a type-and-effect system supporting *effect polymorphism*, that local reasoning principles are restored by lexical effect handlers, using a logical-relations argument. Effect polymorphism, while syntactically heavier-weight than second-class values used by Zhang et al. [35], allows more programs to be expressed.

Biernacki et al. [4] coin the term *lexically scoped handlers* and study two semantics for them: *open* and *generative*. They show that generativity is necessary when effect operations can be polymorphic. Existing languages supporting lexical handlers, including Lexa, freshly generate labels for each installed handler instance at run time.

This generativity is also seen with multi-prompt delimited control [13]. Indeed, StackTrek is similar to libmprompt [17], a C/C++ library for multi-prompt delimited control. Both libraries support constant-time stack switching. StackTrek supports additional optimizations, such as avoiding the allocation of new stacks when the delimited continuation is not used or is invoked in a tail position.

Brachthäuser et al. [7] present Effekt's type-and-effect system for lexically scoped handlers. It features lightweight effect polymorphism via second-class values, as also seen in Zhang et al. [35], and translates it to System Ξ, a calculus where handlers are passed explicitly. System Ξ is then translated to a region-based calculus Λ_{cap}; the translation is known as lift inference [22]. Λ_{cap} is further compiled to System F [27]. System Ξ is similar to Lexa, in that both are intermediate languages where handlers are passed explicitly. It is different in that System Ξ is typed and imposes the second-class restriction on functions, handlers, and continuations, which simplifies lift inference. System C [6] extends System Ξ with first-class functions, but how to translate System C to Λ_{cap} is left as future work. Other than lift inference, Effekt has implementations that target a monad for multi-prompt delimited control [9, 8]. As discussed in Section 2, both implementation strategies involve run-time computations to lift a raised effect out of the extents of its dynamically enclosing handlers until the right handler is found.

WasmFX is a proposal for adding effect handlers to WebAssembly [23]. The proposal is largely based on dynamically scoped handlers but does consider the possibility of *named handlers*, which can be considered as a form of lexical handlers and are given a generative semantics.

Xie et al. [31] present a type-and-effect system for first-class named handlers in Koka. Koka is initially designed around dynamically scoped handlers [16], and Xie and Leijen [32] formalize the compilation of dynamically scoped handlers to a language with multi-prompt delimited control, passing *evidence vectors* that can be looked up to find the correct handler. The implementation of named handlers in Koka is largely based on the same mechanism, also targeting multi-prompt delimited control while passing evidence as first-class values. The Lexa compiler targets a lower-level language, manipulating stack layout directly.

Liu et al. [19] introduce *interruptible iterators* in the context of the JMatch language. Two back ends are presented: a CPS translation to Java and a direct translation to C++. Zhang et al. [36] introduce *bidirectional effects*, subsuming interruptible iterators and generalizing formalisms of effect handlers. Applications beyond iterators are presented, such as async–await with exceptions. A semantic soundness result is established, while an implementation is left as future work. Our formalization and implementation of Lexa support bidirectional effects. It is also possible to encode bidirectional effects by passing thunked effectful computations to resumptions [36].

10 Conclusion

We have presented an approach to compiling lexical effect handlers to low-level stack switching. The compilation is faithful to the lexical scoping discipline, eliminating the cost for run-time search for handlers. Our implementation is guided by a formal model of translation that is proven to be semantics-preserving. The upshot is that the lexical-scoping semantics of effect handlers not only affords local reasoning principles, as previously established, but also enables good performance, as suggested by our empirical results. We hope that this work will encourage language designers and implementers to explore lexical scoping as a viable alternative to dynamically scoped effect handlers.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We thank Jonathan Brachthäuser and Charles Jian for valuable discussions and 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 [20]. The latest release of the Lexa compiler can be found at the following link:

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

References

[1] [n.d.]. Effect handlers benchmarks suite. https://github.com/effect-handlers/effect-handlers-bench Accessed: 2024-04-01.

[2] [n.d.]. MLton. http://mlton.org

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. of the ACM on Programming Languages (PACMPL)* 2, POPL (Jan. 2018). https://doi.org/10.1145/3158096

[4] 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

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988). https://doi.org/10.1002/spe.4380180902

[6] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. of the ACM on Programming Languages (PACMPL)* 6, OOPSLA1 (April 2022). https://doi.org/10.1145/3527320

[7] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. 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

[8] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. 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

[9] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming (JFP)* 17, 6 (2007). https://doi.org/10.1017/S0956796807006259

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

[11] 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

[12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2024. *The Java Language Specification* (SE 22 ed.). Oracle America, Inc. https://docs.oracle.com/javase/specs/jls/se22/jls22.pdf

[13] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA).*

[14] Daniel Hillerström. [n.d.]. multicont. https://opam.ocaml.org/packages/multicont Accessed: 2024-07-01.

[15] 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

[16] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL).*

[17] Daan Leijen and KC Sivamarakrishnan. 2021. libmprompt. https://github.com/koka-lang/libmprompt

[18] 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

[19] Jed Liu, Aaron Kimball, and Andrew C. Myers. 2006. Interruptible iterators. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1111037.1111063

[20] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. *Lexical Effect Handlers, Directly (artifact)*. https://doi.org/10.5281/zenodo.13770453

[21] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. *Lexical Effect Handlers, Directly (Extended Version)*. Technical Report CS-2024-04. School of Computer Science, University of Waterloo.

[22] 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

[23] 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

[24] Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (Feb. 2003).

[25] Gordon Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013).

[26] Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style. In *ACM Conf. on LISP and Functional Programming*. https://doi.org/10.1145/141471.141563

[27] 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

[28] 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

[29] 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

[30] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. of the ACM on Programming Languages (PACMPL)* 4, ICFP (2020). https://doi.org/10.1145/3408981

[31] 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

[32] Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c. *Proc. of the ACM on Programming Languages (PACMPL)* 5, ICFP (2021). https://doi.org/10.1145/3473576

[33] 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

[34] 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

[35] 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

[36] 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 2024-04-06; accepted 2024-08-18