Lexical Effect Handlers, Directly (Extended Version)

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 promising effect-handling 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 of lexical effect handlers could potentially induce a cost similar to that of a run-time search for dynamically scoped handlers. This paper shows how lexical effect handlers can be compiled in a direct style, following the lexical scoping principle and targeting a low-level language that allows control over stack layout. Core aspects of this compilation are formalized and proven correct, and a prototype compiler is implemented. Evaluation on a suite of benchmarks suggests that the compilation strategy generates code of competitive performance, in some cases reducing quadratic increase in running time to linear.

1 INTRODUCTION

Effect handlers [17, 18] 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 [27, 3, 26].

In response to these challenges, *lexical effect handlers* have emerged as a promising design by lexically scoping effect handlers [27, 26, 4, 5, 20]. A handler acts as a lexically scoped capability: only code holding the capability (i.e., code within the lexical scope of the handler) 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 [27] for compiling lexically scoped exception handlers to Java's exception handlers, which are dynamically scoped [8]. 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 [20, 15]. Effekt performs an iterated continuation-passing style (CPS) transformation to compile an intermediate language with lexical effect handlers to an ordinary functional language without handlers. 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. It is claimed 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 and, more importantly, can be a source of inefficiency.

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* style that is faithful to the lexical scoping discipline, thus truly eliminating the need for the run-time search for handlers?

Our idea is a simple 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, no run-time search is needed to identify the code address to branch to. In addition, the suspended computation (aka resumption) is captured directly as stacks or stack frames without search.

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: Lexi, an intermediate language with lexical effect handlers, and Salt, an assembly-like language with control over stack layout. Section 4 describes the translation from Lexi to Salt, Section 5 extends the translation to address tail-resumptive and abortive handlers, and Section 6 establishes the correctness of the translation. Section 7 describes a prototype compiler, lexic. Section 8 presents an empirical evaluation, comparing lexic 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 [21], 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 effects [21]. It would be straightforward to define a type-and-effect system for lexical handlers; the problem is well studied [27, 26, 4, 5, 28] 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` runs 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).

Intuitively, raising `yield` suspends the current job and hands control back to the scheduler, while 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 and 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 [10]: 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`

```
 1 (* library code *)
 2 effect Process =
 3 | yield : 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 ()
```

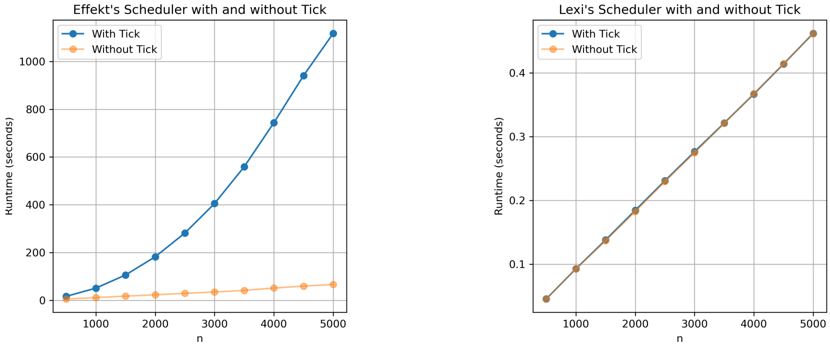
Figure 1. Lightweight cooperative multitasking via lexical effect handlers.

(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; a lighter-weight syntax is possible by allowing omitted handler annotations to be resolved to handler bindings in the lexical context [27, 26, 5].

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.

Performance. It is believed that Effekt [5, 15, 20], a language that most prominently features lexical effect handlers, does not require a run-time search for handlers when an effect is raised. So



(a) Effekt

(b) Lexi

Figure 2. Scaling plots for the scheduler program implemented in Effekt and in Lexi

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 profile of the program!

Why are the `Tick` effects expensive for this scheduler implemented in Effekt? The Effekt compiler statically computes lifting information that, at run time, effectively computes how many handlers in the surrounding evaluation context need to be skipped to find the right handler for an effect. This lifting information is reified as a series of function applications, which is then executed at run time. Hence, although there is no run-time search of the call stack for handlers—there is not a call stack to begin with, as Effekt CPS-transforms programs—using the compiled lifting information incurs a cost similar to that required of a stack walk.

For 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 is $O(n^2)$.

While lifting is perhaps necessary for a CPS compiler like Effekt, 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, lifting goes against the grain of lexical handlers. Lifting, as a language mechanism, is introduced by Biernacki et al. [3] to tame dynamically scoped handlers so that they do not break abstraction safety. The abstraction-safety problem does not arise in the context of lexical handlers [26], so lifting ought to be unnecessary! This paper offers a strategy to compile lexical handlers without lifting. As Figure 2 shows, our compiler implementation generates code that exhibits linear running time, with and without `Tick` effects.

3 TWO CORE LANGUAGES: LEXI AND SALT

We want to formalize the key aspects of the compilation of lexical effect handlers to a stack-based low-level language. To this end, this section defines two core languages, Lexi and Salt. Sections 4–6 will define the translation between them and prove the correctness of the translation. Lexi is a

$$
\begin{array}{llll}
\text{constant} & c & ::= & i \mid \mathfrak{L} \mid L \mid \mathtt{ns} \\
\text{value} & v & ::= & x \mid c \\
\text{heap value} & V & ::= & \langle v_1, \cdots, v_n \rangle \mid \mathtt{cont}\ K \\
\text{expression} & e & ::= & v \mid v_1 + v_2 \mid \mathtt{newref}\ V \mid \pi_i\,(v) \mid v_1\,[v_2] \leftarrow v_3 \mid v(v_1, \cdots, v_n) \mid \\
 & & & \mathtt{handle}\ \mathfrak{L}_{\text{body}}\ \mathtt{with}\ \mathfrak{L}_{\text{op}}\ \mathtt{under}\ v_{\text{env}} \mid \mathtt{raise}\ v_1\ v_2 \mid \mathtt{resume}\ v_1\ v_2 \mid \\
 & & & \mathtt{exit}\ v \\
\text{term} & t & ::= & v\ \mathtt{end} \mid \mathtt{let}\ x = e\ \mathtt{in}\ t \mid \mathtt{halt}\ c \\
\text{program} & P & ::= & \mathtt{letrec}\ \mathfrak{L}_1 \mapsto \lambda\overline{x}.t, \cdots, \mathfrak{L}_n \mapsto \lambda\overline{x}.t\ \mathtt{in}\ \mathfrak{L}_{\text{main}}() \\
\text{local environment} & E & ::= & [x_1 \to v_1, \cdots, x_n \to v_n] \\
\text{code memory} & \mathfrak{H} & ::= & [\mathfrak{L}_1 \to \lambda\overline{x}.t, \cdots, \mathfrak{L}_n \to \lambda\overline{x}.t] \\
\text{heap} & H & ::= & [L_1 \to V_1, \cdots, L_n \to V_n] \\
\text{frame} & F & ::= & (E, \mathtt{let}\ x = \square\ \mathtt{in}\ t) \mid \#L^{\mathfrak{L}_{\text{op}}, L_{\text{env}}}\square \\
\text{evaluation context} & K & ::= & \square \mid K \cdot F \\
\text{configuration} & M & ::= & \langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel t \rangle
\end{array}
$$

Figure 3. Syntax of Lexi

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 Lexi and Salt to capture the core ideas of how we compile lexical handlers; it is not our goal to define a formally verified, realistic compiler. The actual 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 interpreters for both languages and the translation; they can be found in the supplementary material.

3.1 Source Language: Lexi

Lexi programs use lexically scoped variables to identify handlers. Lexi 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 resolving uses of handlers to variables in scope, as described and formalized in prior work [26, 5]. Programs in Lexi are assumed to have undergone closure conversion and hoisting, so that all functions, including those representing handlers, are closed and lifted to the top level.

Lexi features important expressive power not found in System Ξ [5], an intermediate language used by the Effekt compiler. System Ξ requires that functions, resumptions, and handlers be second-class (that is, they cannot be returned or stored in data structures). Effekt's lift inference is said to be greatly simplified by these restrictions [15]. Lexi does not impose these restrictions. In addition, Lexi enables bidirectional effects [28], namely the ability for an effect handler to raise reverse-direction effects to the computation that raised the initial effect.

Like MultiCore OCaml [21], which also compiles to call stacks, it is not our goal to support handlers with multishot resumptions. Thus, in the Lexi formalism, a resumption can be resumed at most once. However, our compiler implementation does have limited support for multishot resumptions.

As an intermediate language, Lexi is untyped, but it would be straightforward to extend Lexi with a type-and-effect system to ensure type safety, effect safety, and abstraction safety [26].

Syntax. Figure 3 presents the syntax of Lexi. *Terms* are in A-normal form [19, 7]. A term `let` $x_1 = e_1$ `in` $\cdots$ `let` $x_n = e_n$ `in` v end binds expressions to variables and ending with a value. We will sometimes elide the trailing end for brevity. A term is evaluated under a code memory $\mathfrak{H}$, a data memory (aka heap) H, an evaluation context K, and a local variable environment E.

Metavariable $\mathfrak{L}$ ranges over *code labels*, which exist in the program text. 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 *value* is either a local variable x, an integer i, a code label $\mathfrak{L}$, a data label L, or `ns` (nonsense). Expressions and terms evaluate to values.

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 *expression* takes the form of a value, an arithmetic operation, allocating a tuple, reading and writing a tuple, applying a function, installing a handler, raising an effect to an installed handler, and resuming a resumption.

In `handle` $\mathfrak{L}_{\text{body}}$ `with` $\mathfrak{L}_{\text{op}}$ `under` v_{env}, $\mathfrak{L}_{\text{op}}$ is the label of the handler code (i.e., the implementation of the effect operation), $\mathfrak{L}_{\text{body}}$ 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 $\mathfrak{L}_{\text{op}}$ and $\mathfrak{L}_{\text{body}}$. The environment is passed to the closures as an argument when they are called. 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 compiler implementation is free of these restrictions.

A Lexi *program* is a sequence of top-level, possibly mutually recursive functions, one of which is considered the main function. A code memory $\mathfrak{H}$ maps code labels to functions. Evaluation of a program starts by loading the top-level functions into the code memory.

Operational semantics. We give an operational semantics to Lexi as an abstract machine. The full set of reduction rules can be found in an appendix. Figure 4 shows selected rules.

As alluded to earlier, the machine state consists of an immutable code memory, a mutable data heap, an evaluation context, a local environment, and a term under evaluation. A local environment E records the values of arguments and local variables for the activation of a function. An evaluation context K is a sequence of *frames* F. A frame can be an *activation frame* (E, `let` $x = \square$ `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* $\#L^{\mathfrak{L}_{\text{op}}, L_{\text{env}}}\square$, which consists of a label L identifying the handler instance, a label $\mathfrak{L}_{\text{op}}$ identifying the handler code, a label L_{env} identifying the closure environment, and a hole $\square$ where evaluation is in progress.

To run a program, the initial configuration is constructed by moving the function definitions into the code memory and invoking the main function. 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, as an example, the `NEW` rule in Figure 4 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.

NEW

$$\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{newref}\ \langle v_1, \cdots, v_n \rangle\ \texttt{in}\ t \rangle \longrightarrow$$
$$\left\langle \mathfrak{H} \parallel H[L \mapsto \left\langle \hat{E}(v_1), \cdots, \hat{E}(v_n) \right\rangle] \parallel K \parallel E[x \mapsto L] \parallel t \right\rangle$$

where L is fresh

HANDLE

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

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

LEAVE

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

RAISE

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

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

RESUME

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

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

Figure 4. Selected reduction rules of Lexi.

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.

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 $\mathfrak{L}_{\text{body}}$ receives: L_{env} is the closure environment of $\mathfrak{L}_{\text{body}}$, and L identifies the handler instance newly pushed onto the evaluation context. The body of the function $\mathfrak{L}_{\text{body}}$ then becomes the term to be evaluated next. The generativity of the handler label L matches the semantics of previous languages supporting lexical handlers [27, 5].

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 first evaluates the operands v_1 and v_2 of `raise`: v_1 evaluates to the handler label L of the handler instance to raise to, and v_2 evaluates to the argument of the effect operation. A handler frame matching the label L is found in the evaluation context, and the evaluation context is cut to

location	ℓ	$::=$	$\mathfrak{L} \mid L \mid \texttt{next}(\ell)$
word	v	$::=$	$\ell \mid i \mid \texttt{ns}$
operand	o	$::=$	$\mathbf{r} \mid v$
stack	s	$::=$	$\texttt{nil} \mid s :: v$
heap value	V	$::=$	$\langle v_1, \cdots, v_n \rangle \mid s$
code memory	$\mathfrak{H}$	$::=$	$\{\mathfrak{L}_1 \mapsto I_1, \cdots, \mathfrak{L}_n \mapsto I_n\}$
heap	H	$::=$	$\{L_1 \mapsto V_1, \cdots, L_n \mapsto V_n\}$
register file	R	$::=$	$\{\mathbf{sp} \mapsto v_{\mathrm{sp}}, \mathbf{ip} \mapsto v_{\mathrm{ip}}, \mathbf{r1} \mapsto v_1, \cdots, \mathbf{rn} \mapsto v_n\}$
instruction	ι	$::=$	$\texttt{add}\ \mathbf{r}, o \mid \texttt{mkstk}\ \mathbf{r} \mid \texttt{salloc}\ i \mid \texttt{sfree}\ i \mid \texttt{malloc}\ \mathbf{r}_d, i \mid$
			$\texttt{mov}\ \mathbf{r}_d, o \mid \texttt{load}\ \mathbf{r}_d, [\mathbf{r}_s + i] \mid \texttt{store}\ [\mathbf{r}_d + i], \mathbf{r}_s \mid$
			$\texttt{push}\ o \mid \texttt{pop}\ \mathbf{r} \mid \texttt{call}\ o \mid \texttt{halt}$
instruction sequence	I	$::=$	$\iota; I \mid \texttt{return} \mid \texttt{jmp}$
configuration	M	$::=$	$\langle \mathfrak{H} \parallel H \parallel R \rangle$
program	P	$::=$	$\{\mathfrak{L}_1 \mapsto I_1, \cdots, \mathfrak{L}_n \mapsto I_n\}$

Figure 5. Syntax of Salt

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

The RESUME rule evaluates the operands v_1 and v_2 of `resume`: v_1 evaluates to a label L_k identifying the resumption, and v_2 evaluates to the argument. 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 multiple stacks and low-level control over stack layout.

Syntax. Figure 5 presents the syntax 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 $\mathfrak{L}$. 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 (i.e., $H(L) = \texttt{nil} :: v_1 :: \cdots :: v_n$), then $L^j = \texttt{next}^j(L)$ is the stack location where v_j is stored ($j \leq n$).

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

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

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

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

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

Figure 6. Selected reduction rules 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 $\mathbf{r}$. 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* $\mathfrak{H}$ maps locations $\mathfrak{L}$ to instruction sequences. An *instruction sequence* I ends with a return or jmp. If $\mathfrak{H}(\mathfrak{L}) = \iota_0; \cdots; \iota_i; I$, then the notations $\texttt{next}^i(\mathfrak{L})$ and $\mathfrak{L}^i$ mean the address of the start of the instruction subsequence $\iota_i; I$. We define a meta-level partial function $\hat{\mathfrak{H}}(\ell)$ such that $\hat{\mathfrak{H}}(\mathfrak{L}^i) = \iota_i$ if $\mathfrak{H}(\mathfrak{L}) = \iota_0; \cdots; \iota_i; I$ and $\hat{\mathfrak{H}}(\mathfrak{L}^i) = \texttt{return}$ if $\mathfrak{H}(\mathfrak{L}) = \iota_0; \cdots; \iota_{i-1}; \texttt{return}$ (similarly for jmp).

A *program* is a finite map from code addresses to instruction sequences (i.e., top-level functions). One of the code addresses is considered the main function.

Operational semantics. The operational semantics of Salt is standard of an abstract assembly language. The full set of rules can be found in an appendix. Figure 6 shows selected rules that deal with stacks. The machine state consists of an immutable code memory, a mutable heap, and a mutable register file. Reduction takes the form $\langle \mathfrak{H} \parallel H \parallel R\rangle \longrightarrow \langle \mathfrak{H} \parallel H' \parallel R'\rangle$.

To run a program, the initial configuration is constructed as follows: load into the code memory all top-level functions and also an entry-point function $\mathcal{L}_{\text{init}} \mapsto \texttt{call}\ \mathcal{L}_{\text{main}}; \texttt{halt}$ that calls the main function, allocate a stack $L \mapsto \texttt{nil}$ on the heap, make **sp** point to L, and make **ip** point to $\mathfrak{L}_{\text{init}}$. The program terminates successfully when **ip** points to a halt instruction.

To take a step, the instruction at the address stored in **ip** is executed. 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 locations 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 stack top L^m. The opcodes call and return are like those in x86: they push or pop the return address to 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, which distinguishes our approach from previous works [23, 25, 20] that define translations to formal models of high-level functional languages à la System F.

4 TRANSLATING LEXI TO SALT

The translation from Lexi to Salt is defined with functions $\llbracket \lambda\overline{x}.\, t \rrbracket = I$, $\llbracket t \rrbracket_{\Gamma} = ▯$, $\llbracket e \rrbracket_{\Gamma} = ▯$, and $\llbracket v \rrbracket_{\Gamma}^{\mathbf{r}} = \iota$, for Lexi functions, terms, expressions, and values, respectively. For visual clarity, Lexi constructs are typeset in blue and Salt constructs in red. The metavariable $▯ ::= \epsilon \mid \iota; ▯$ denotes a sequence of Salt instructions that, unlike I, does *not* end with a `return` or `jmp`. The metavariable $\Gamma ::= \epsilon \mid \Gamma, x$ denotes a sequence of local variables in Lexi.

Figure 8 defines the translation for selected Lexi constructs. A full version can be found in an appendix. $\llbracket \lambda\overline{x}.\, t \rrbracket = I$ translates a Lexi 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.

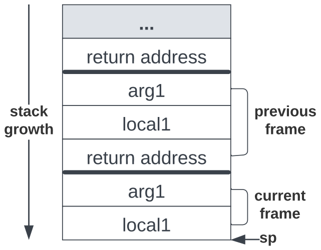


Figure 7. Stack layout in Salt

The layout of a single stack is shown in Figure 7. The stack grows downwards. 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 last frame does not have a return address.

$\llbracket t \rrbracket_{\Gamma} = ▯$ translates a term t, with Γ providing bindings for the free variables in t. The result of the term t is left in **r1**. $\llbracket e \rrbracket_{\Gamma} = ▯$ translates an expression e, putting the result in **r1**. Most cases of this translation are ordinary, except for `handle`, `raise`, and `resume` expressions. $\llbracket v \rrbracket_{\Gamma}^{\mathbf{r}} = \iota$ translates a value v to an instruction ι that stores the word in the provided register **r**. Notice that, although data labels L are first-class values in Lexi, they cannot appear in Lexi program text (they must be freshly generated), the translation is not defined for them.

We now illustrate the translation of `handle`, `raise`, and `resume` expressions. Most of the work is done with the built-in trampoline functions $\mathfrak{L}_{\text{handle}}$, $\mathfrak{L}_{\text{raise}}$, and $\mathfrak{L}_{\text{resume}}$, which are provided in the right column of Figure 8 as a reference.

We will use the scheduler program in Figure 1 as a running example. Figure 9 shows the layout of stacks at different points of the execution of this running example, which we will use to guide the discussion.

$\boxed{\llbracket \lambda \overline{x}.\, t \rrbracket = I}$

$\llbracket \lambda(x_1, \cdots, x_n).\, t \rrbracket =$

$\quad \texttt{push } \mathbf{rn};\ \cdots;\ \texttt{push } \mathbf{r1};$

$\quad \llbracket t \rrbracket_{x_n, \cdots, x_1};\ \texttt{sfree } k;\ \texttt{return}$

$\quad \text{where } k = n + \text{NUM\_LET}(t)$

$\boxed{\llbracket t \rrbracket_\Gamma = ▯}$

$\llbracket \texttt{let } x = e \texttt{ in } t \rrbracket_\Gamma = \llbracket e \rrbracket_\Gamma;\ \texttt{push } \mathbf{r1};\ \llbracket t \rrbracket_{\Gamma, x}$

$\llbracket v \texttt{ end} \rrbracket_\Gamma = \llbracket v \rrbracket_\Gamma$

$\boxed{\llbracket e \rrbracket_\Gamma = ▯}$

$\llbracket v \rrbracket_\Gamma = \llbracket v \rrbracket_\Gamma^{\mathbf{r1}}$

$\llbracket v_0(v_1, \cdots, v_n) \rrbracket_\Gamma =$

$\quad \llbracket v_n \rrbracket_\Gamma^{\mathbf{rn}};\ \cdots;\ \llbracket v_1 \rrbracket_\Gamma^{\mathbf{r1}};\ \llbracket v_0 \rrbracket_\Gamma^{\mathbf{r0}};$

$\quad \texttt{call } \mathbf{r0}$

$\llbracket \texttt{handle } \mathfrak{L}_{\text{body}} \texttt{ with } A\ \mathfrak{L}_{\text{op}} \texttt{ under } v_{\text{env}} \rrbracket_\Gamma =$

$\quad \llbracket A \rrbracket^{\mathbf{r4}};\ \llbracket v_{\text{env}} \rrbracket_\Gamma^{\mathbf{r3}};$

$\quad \texttt{mov } \mathbf{r2}, \mathfrak{L}_{\text{op}};$

$\quad \texttt{mov } \mathbf{r1}, \mathfrak{L}_{\text{body}};$

$\quad \texttt{call } \mathfrak{L}_{\text{handle}}$

$\llbracket \texttt{raise } v_1\ v_2 \rrbracket_\Gamma =$

$\quad \llbracket v_2 \rrbracket_\Gamma^{\mathbf{r2}};\ \llbracket v_1 \rrbracket_\Gamma^{\mathbf{r1}};\ \texttt{call } \mathfrak{L}_{\text{raise}}$

$\llbracket \texttt{resume } v_1\ v_2 \rrbracket_\Gamma =$

$\quad \llbracket v_2 \rrbracket_\Gamma^{\mathbf{r2}};\ \llbracket v_1 \rrbracket_\Gamma^{\mathbf{r1}};\ \texttt{call } \mathfrak{L}_{\text{resume}}$

$\boxed{\llbracket v \rrbracket_\Gamma^{\mathbf{r}} = \iota}$

$\llbracket x_i \rrbracket_{x_n, \cdots, x_0}^{\mathbf{r}} = \texttt{load } \mathbf{r}, [\mathbf{sp} + i]$

$\llbracket i \rrbracket_\Gamma^{\mathbf{r}} = \texttt{mov } \mathbf{r}, i$

$\llbracket \mathfrak{L} \rrbracket_\Gamma^{\mathbf{r}} = \texttt{mov } \mathbf{r}, \mathfrak{L}$

$\boxed{\llbracket A \rrbracket^{\mathbf{r}} = \iota}$

$\llbracket \texttt{general} \rrbracket^{\mathbf{r}} = \texttt{mov } \mathbf{r},\ 0$

$\llbracket \texttt{tail} \rrbracket^{\mathbf{r}} = \texttt{mov } \mathbf{r},\ 1$

$\llbracket \texttt{abort} \rrbracket^{\mathbf{r}} = \texttt{mov } \mathbf{r},\ 2$

```
// r1: $\mathfrak{L}_{body}$, r2: $\mathfrak{L}_{op}$, r3: $L_{env}$, r4 : A
$\mathfrak{L}_{\text{handle}}$ ↦
  mov r5, sp;     // save old stack pointer
  mkstk sp;     // create new stack
  push r2; push r3; push r4;
  push r5;     // create header frame
  mov r6, r1;     // r6 stores $\mathfrak{L}_{body}$
  mov r2, sp;     // r2 points to header frame
  mov r1, r3;     // r1 points to environment tuple
  call r6;     // call $\mathfrak{L}_{body}$ with args in r1, r2
  pop r2;     // r2 points to parent stack
  sfree 3;     // deallocate rest of header frame
  mov sp, r2;     // switch to parent stack
  return

// r1: pointer to header frame, r2: op arg
$\mathfrak{L}_{\text{raise}}$ ↦
  load r4, [r1 + 0];     // r4 points to top of handler's stack
  store [r1 + 0], sp;     // save resumption's sp in exchanger
  mov sp, r4;     // switch to handler's stack
  load r5, [r1 + 2];     // r5 stores $\mathfrak{L}_{op}$
  malloc r3, 1;
  store [r3 + 0], r1;     // r3 points to resumption object
  load r1, [r1 + 1];     // r1 points to environment tuple
  call r5;     // call $\mathfrak{L}_{op}$ with args in r1, r2, r3
  return

// r1: pointer to resumption object, r2: resumption arg
$\mathfrak{L}_{\text{resume}}$ ↦
  load r3, [r1 + 0];     // r3 points to resumption header frame
  store [r1 + 0], ns;     // invalidate resumption object
  load r4, [r3 + 0];     // r4 points to resumption's sp
  store [r3 + 0], sp;     // exchanger points to handler's stack
  mov r1, r2;     // r1 stores resumption arg
  mov sp, r4;     // switch to resumption's stack
  return
```

Figure 8. Left: Translation rules for selected constructs. Right: Built-in functions called by the translated code.

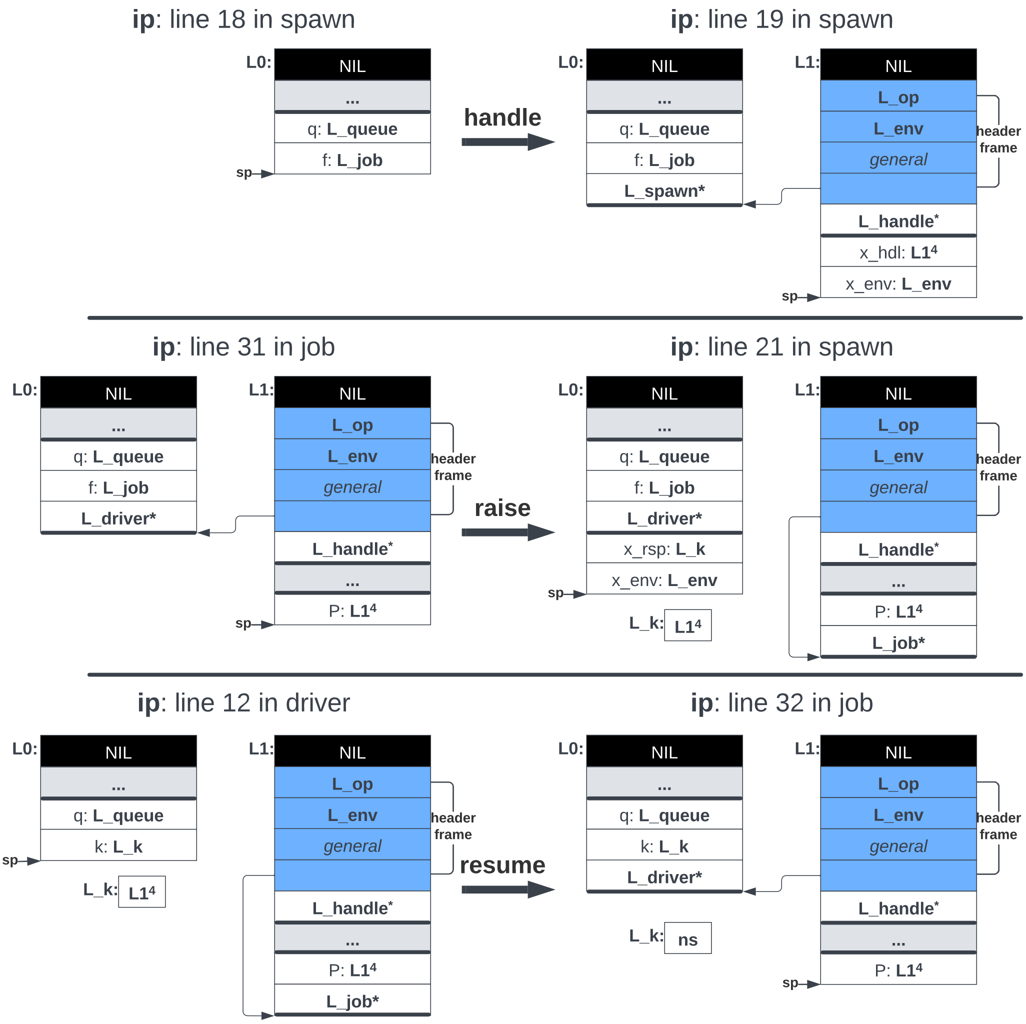


Figure 9. Stack layout in Salt of the running example in different states of execution. **Top row:** before and after entering the `handle` expression at line 18. **Middle row:** before and after raising at line 31. **Bottom row:** before and after resuming at line 12.

Entering a `handle` expression. The two stack layouts on the first row of Figure 9 are taken immediately before and after entering the `handle` expression (line 18). In the snapshot on the left, the stack pointer **sp** points to the top of the stack `L0`. Two arguments of the function `spawn` are on the current frame. The translation of the `handle` expression calls the built-in function $\mathfrak{L}_{\text{handle}}$, which pushes the return address $\mathfrak{L}^*_{\text{spawn}}$ (`*` denotes an offset that is irrelevant to the discussion) to the current stack. The control is then transferred to $\mathfrak{L}_{\text{handle}}$, which allocates a new stack `L1` with a `mkstk` instruction. It then sequentially pushes four values onto the new stack: the address $\mathfrak{L}_{\text{op}}$ of the handler code, the address L_{env} of the closure environment, the handler annotation (to be discussed in Section 5), and the old value of **sp** before the stack switch. 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`.

After the header frame is initialized, the control is transferred to the body of the `handle` clause (line 19). Two arguments are received by the handled body: the closure environment L_{env} and the location of the newly installed handler `L1`^{4}. Notice that in the running example, the `handle` clause is an expression rather than a top-level function, but we assume that standard compiler passes (such as closure conversion, lambda hoisting, and A-normalization) exist to transform the program into the Lexi syntax.

Raising an effect and capturing the resumption. The snapshots in the second row show the stack layout immediately before and after the `raise` at line 31 in the running example. The return address—that is, the program point where the resumption will pick up—is pushed to the current stack (`L1`) first. Next, the current stack pointer is exchanged with the one in the exchanger (accessed through the local variable `P`), so that **sp** points to the top of the parent stack (`L0`). Notice how the lexically scoped local variable `P` is used to identify the handler to which the effect is raised.

A single-element tuple is then allocated to store the address `L1`^{4} of the exchanger. The address of this single-element tuple, `L_k`, serves as the identity of the resumption. This address `L_k`, along with the address of the handler closure environment, are pushed to the parent stack (`L0`) as the arguments of the call to the handler code `L_op`. Control is subsequently transferred to `L_op`, which handles the raised effect.

Importantly, no run-time search is needed to identify the handler or the resumption during the raise of an effect.

Resuming a resumption. The stack layout on the third row shows the state of the stacks immediately before and after the `resume` at line 12 in the running example. The return address $\mathfrak{L}^*_{\text{driver}}$ is pushed to the current stack (`L0`) first. Next, the current stack pointer is exchanged with the one in the exchanger (accessed through the resumption variable `k`), so that **sp** points to the top of the resumption's stack (`L1`). The single-element tuple `L_k`, which has been pointing to the exchanger, is updated to `ns`, to prevent the resumption from being resumed again. Control then resumes what was left off by the `raise` at line 32.

5 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 tail position, and an *abortive* handler does not resume the resumption at all. Many important handlers are tail-resumptive (e.g., generator handlers) or abortive (e.g., exception handlers). So it is worthwhile to optimize the compilation of these handlers to make them as efficient as possible.

Syntax. To capture the essence of how these special handlers are translated, we update the syntax of Lexi, as Figure 10 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. This, however, this does not restrict expressiveness, because choosing the right annotation for a `raise` expression can be done at run time by inspecting the annotation of the handler to raise to. Indeed, our implementation does not require the annotation to be statically known for a `raise` expression.

Operational semantics. The HANDLE rule in Figure 4 still works for all three kinds of handlers; it only needs to be updated to include a metavariable A. The RAISE rule in Figure 4 is updated to work only for `raise general` expressions. Two additional RAISE rules are added, as Figure 10 shows. The TAILRAISE rule reduces a `raise tail` expression in place as if it is a regular function call. The ABORTRAISE rule reduces a `raise abort` expression rule by aborting the computation delimited by the handler.

Notice that in both cases, the handler implementation $\mathfrak{H}(\mathfrak{L}_{\text{op}})$ is no longer parameterized by the resumption continuation k; it is implied if or how k is used. Also notice that neither case reifies the resumption as a heap-allocated object, which justifies the optimization of not allocating new stacks.

Translation. The translation of tail-resumptive and abortive handlers is optimized so that the header frame is allocated on the same stack as the parent's. Instead of $\mathfrak{L}_{\text{handle}}$, we use the trampoline $\mathfrak{L}_{\text{handle\_special}}$ when entering tail-resumptive and abortive handlers. As the calling sequence in Figure 10 shows, the trampoline does not use any `mkstk` instructions. Notice that the header frame no longer stores an exchanger, as the exchanger is needed only when the resumption has to be reified.

Figure 10 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 by updating **sp** to point to the abortive handler's header frame and then jumps to the handler's code.

6 CORRECTNESS OF THE FORMALIZED TRANSLATION

To show that the translation is semantics-preserving, we give a simulation proof that relates the execution of a program in Lexi to the execution of the translated program in Salt. Our theoretical framework strictly follows Leroy [12].

We first define two predicates, initial and final, for both languages. The initial(P, M) predicate relates a program P with its initial configuration M, and the final(M, i) predicate relates a terminal configuration M with the result i of the program.

Definition 1 (Initial and final configurations). The predicates $\text{initial}_{\text{Lexi}}(P, M)$, $\text{initial}_{\text{Salt}}(P, M)$, $\text{final}_{\text{Lexi}}(M, i)$, and $\text{final}_{\text{Salt}}(M, i)$ are defined as follows:

- $\text{initial}_{\text{Lexi}}\left(\texttt{letrec}\ \overline{\mathfrak{L}_i \mapsto \lambda\overline{x}.t_i}\ \texttt{in}\ \mathfrak{L}_{\text{main}}(),\ \left\langle\left\{\overline{\mathfrak{L}_i \mapsto \lambda\overline{x}.t_i}\right\} \parallel \{\} \parallel \square \parallel [] \parallel \texttt{let}\ x = \mathfrak{L}_{\text{main}}(),\ \_ = \texttt{exit}\ x\ \texttt{in ns end}\right\rangle\right)$
- $\text{initial}_{\text{Salt}}\left(\left\{\overline{\mathfrak{L}_i \mapsto I_i}\right\},\ \left\langle\left\{\overline{\mathfrak{L}_i \mapsto I_i},\ \mathfrak{L}_{\text{init}} \mapsto \texttt{call}\ \mathfrak{L}_{\text{main}}; \texttt{halt}\right\} \parallel \{L_{\text{stack}} \mapsto \texttt{nil}\} \parallel \left\{\mathbf{sp} \mapsto L^0_{\text{stack}}, \mathbf{ip} \mapsto \mathfrak{L}^0_{\text{init}}\right\}\right\rangle\right)$
- $\text{final}_{\text{Lexi}}\left(\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel \texttt{halt}\ i\rangle, i\right)$
- $\text{final}_{\text{Salt}}\left(\langle \mathfrak{H} \parallel H \parallel R[\mathbf{ip} \mapsto \ell, \mathbf{r1} \mapsto i]\rangle, i\right)$ where $\hat{\mathfrak{H}}(\ell) = \texttt{halt}$

$$\begin{array}{llll}
\text{handler annotation} & A & ::= & \texttt{tail} \mid \texttt{abort} \mid \texttt{general} \\
\text{expression} & e & ::= & \cdots \mid \texttt{handle}\ \mathfrak{L}_{\text{body}}\ \texttt{with}\ A\ \mathfrak{L}_{\text{op}}\ \texttt{under}\ v_{\text{env}} \mid \texttt{raise}\ A\ v_1\ v_2 \\
\text{frame} & F & ::= & \cdots \mid \left(\#L_{A}^{\mathfrak{L}_{\text{op}},L_{\text{env}}}\ \Box\right)
\end{array}$$

TAILRAISE

$$\left\langle \mathfrak{H} \parallel H \parallel K \cdot \left(\#L_{\texttt{tail}}^{\mathfrak{L}_{\text{op}},L_{\text{env}}}\ \Box\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise tail}\ v_1\ v_2\ \texttt{in}\ t \right\rangle \longrightarrow$$
$$\left\langle \mathfrak{H} \parallel H \parallel K \cdot \left(\#L_{\texttt{tail}}^{\mathfrak{L}_{\text{op}},L_{\text{env}}}\ \Box\right) \cdot K' \cdot (E, \texttt{let}\ x = \Box\ \texttt{in}\ t) \parallel \left[x_{\text{env}} \to L_{\text{env}}, y \to \hat{E}(v_2)\right] \parallel t' \right\rangle$$
where $\hat{E}(v_1) = L$ and $\mathfrak{H}(\mathfrak{L}_{\text{op}}) = \lambda(x_{\text{env}}, y).t'$

ABORTRAISE

$$\left\langle \mathfrak{H} \parallel H \parallel K \cdot \left(\#L_{\texttt{abort}}^{\mathfrak{L}_{\text{op}},L_{\text{env}}}\ \Box\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise abort}\ v_1\ v_2\ \texttt{in}\ t \right\rangle \longrightarrow$$
$$\left\langle \mathfrak{H} \parallel H \parallel K \parallel \left[x_{\text{env}} \to L_{\text{env}}, y \to \hat{E}(v_2)\right] \parallel t' \right\rangle$$
where $\hat{E}(v_1) = L$ and $\mathfrak{H}(\mathfrak{L}_{\text{op}}) = \lambda(x_{\text{env}}, y).t'$

$$\begin{array}{l}
\mathfrak{L}_{\text{handle\_special}} \mapsto \\
\quad \texttt{push}\ \mathbf{r2};\ \texttt{push}\ \mathbf{r3};\ \texttt{push}\ \mathbf{r4};\ \texttt{push}\ \texttt{ns}; \\
\quad \texttt{mov}\ \mathbf{r6},\ \mathbf{r1}; \\
\quad \texttt{mov}\ \mathbf{r2},\ \mathbf{sp};\ \ \texttt{mov}\ \mathbf{r1},\ \mathbf{r3}; \\
\quad \texttt{call}\ \mathbf{r6}; \\
\quad \texttt{sfree}\ 4;\ \texttt{return}
\end{array}
\qquad
\begin{array}{l}
[\![\texttt{raise tail}\ v_1\ v_2]\!]_{\Gamma} = \\
\quad [\![v_2]\!]_{\Gamma}^{\mathbf{r2}}\ ;\ [\![v_1]\!]_{\Gamma}^{\mathbf{r1}}\ ; \\
\quad \texttt{load}\ \mathbf{r3},\ [\mathbf{r1}+3]; \\
\quad \texttt{load}\ \mathbf{r1},\ [\mathbf{r1}+2]; \\
\quad \texttt{call}\ \mathbf{r3}
\end{array}
\qquad
\begin{array}{l}
[\![\texttt{raise abort}\ v_1\ v_2]\!]_{\Gamma} = \\
\quad [\![v_2]\!]_{\Gamma}^{\mathbf{r2}}\ ;\ [\![v_1]\!]_{\Gamma}^{\mathbf{r1}}\ ; \\
\quad \texttt{load}\ \mathbf{r3},\ [\mathbf{r1}+3]; \\
\quad \texttt{mov}\ \mathbf{sp},\ \mathbf{r1}; \\
\quad \texttt{load}\ \mathbf{r1},\ [\mathbf{r1}+2]; \\
\quad \texttt{sfree}\ 4;\ \texttt{jmp}\ \mathbf{r3}
\end{array}$$

Figure 10. Extending Lexi and the translation to Salt to address tail-resumptive and abortive handlers. The HANDLE and RAISE rules are omitted as they require minimal changes relative to the rules in Figure 4.

The initial configuration of Lexi is set up as follows: the code memory is initialized to the top-level functions, the heap is empty, the evaluation context is empty, the local environment is empty, and the term is a call to the main function followed by an exit.

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 $\mathfrak{L}_{\text{init}}$ that calls the main function and `halt`. The heap contains an initial empty stack with the base address L_{stack}. In the register file, **sp** points to the initial stack L_{stack}, and **ip** points to the initial instruction $\mathfrak{L}_{\text{init}}$.

A final configuration of Lexi has the term `halt` i, and i is the result of the program.

In a final configuration of Salt, **ip** points to a `halt` instruction. The value of **r1** is read off from the register file as the result of the program.

Next we define the *observable behavior* 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(P, M) relation, a final(M, i) relation, and a reduction relation $\longrightarrow$ for a language, the *observable behavior* B of a program P in the language is defined as follows:

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

$$\boxed{P \Downarrow B}$$

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

With observable behaviors defined, we can define semantic preservation.

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

To prove that our translation from Lexi to Salt preserves semantics, we use a simulation argument: we construct a relation $\sim$ that relates the configurations of Lexi to 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 M_1, M_2, and M_1' such that $M_1 \sim M_2$ and $M_1 \longrightarrow M_1'$, there exists M_2' such that $M_2 \longrightarrow^+ M_2'$ and $M_1' \sim M_2'$.

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 a $\mathcal{L}_1$ program P_1 and its translation $P_2 = [\![P_1]\!]$ in $\mathcal{L}_2$, the following conditions hold:

(1) If $\text{initial}_{\mathcal{L}_1}(P_1, M_1)$ and $\text{initial}_{\mathcal{L}_2}(P_2, M_2)$, then $M_1 \sim M_2$.
(2) If $M_1 \sim M_2$ and $\text{final}_{\mathcal{L}_2}(M_1, i)$, then $\text{final}_{\mathcal{L}_2}(M_2, i)$.
(3) $\longrightarrow$ simulates $\longrightarrow$ with respect to $\sim$.

The proof of Theorem 1 can be found in Leroy [12]. Given the theorem, all we need to do is to construct the $\sim$ relation and show that it satisfies the three conditions.

We now construct the $\sim$ relation between Lexi and Salt. Some definitions are omitted for brevity.

$$\boxed{\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel t \rangle \sim \langle \mathfrak{H} \parallel H \parallel R \rangle}$$

$$\frac{(E, t) \overset{\text{ins}}{\sim} I \qquad (K, E) \underset{\Xi}{\overset{\text{context}}{\sim}} (H_{\text{stack}}, \ell_{\text{sp}}) \qquad H \underset{\Xi}{\overset{\text{data}}{\sim}} H_{\text{heap}} \qquad \mathfrak{H} \underset{\Xi}{\overset{\text{code}}{\sim}} \mathfrak{H} \qquad \mathfrak{H}(\mathfrak{L}) = \iota_0 :: \cdots \iota_j :: I}{\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel t \rangle \sim \left\langle \mathfrak{H} \parallel H_{\text{stack}} \cup H_{\text{heap}} \parallel \left\{\mathbf{sp} \mapsto \ell_{\text{sp}}, \mathbf{ip} \mapsto \mathfrak{L}^{j+1}\right\}\right\rangle}$$

The $\sim$ relation is defined using several auxiliary relations that relate components of the configurations. Some auxiliary relations are indexed by a context Ξ that maps Lexi labels 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 Lexi term t to a Salt instruction sequence I. The relation also takes as input a Lexi 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 $(K, E) \underset{\Xi}{\overset{\text{context}}{\sim}} (H_{\text{stack}}, \ell_{\text{sp}})$ relates the evaluation context K in Lexi to a portion of the heap H_{stack} in Salt. An evaluation context can correspond to multiple stacks in Salt, so H_{stack} 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 $H \underset{\Xi}{\overset{\text{data}}{\sim}} H_{\text{heap}}$ relates the heap H in Lexi to the heap H_{heap} in Salt. The Lexi heap contains tuples and resumptions. How Lexi tuples and Salt tuples are related is straightforward. As for resumptions, one resumption in Lexi corresponds to a single-element tuple and a collection of stacks in Salt. The correspondence between a Lexi resumption and a collection of Salt stacks is similar to the $\underset{\Xi}{\overset{\text{context}}{\sim}}$ relation discussed earlier.

The fourth premise $\mathfrak{H} \underset{\Xi}{\overset{\text{code}}{\sim}} \mathfrak{H}$ relates the code memory $\mathfrak{H}$ in Lexi to the code heap $\mathfrak{H}$ in Salt. The relation is straightforward.

The final premise states that the instruction sequence I, pointed to by the instruction pointer $\mathbf{ip}$, must be a subsequence of a function in the code memory $\mathfrak{H}$. As the Lexi term t reduces, $\mathbf{ip}$ is updated to point to the next instruction that corresponds to the reduced term.

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

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

7 A PROTOTYPICAL COMPILER

The previous sections provide a formal account of the compilation process. In this section, we describe the implementation of our compiler called lexic. lexic compiles a Lexi program into a C program, supported by a C macro library called StackTrek we have developed. The compiled C program is then sent to LLVM for further compilation. The compilation from Lexi to C is syntax-directed and does not involve any optimization; we leave all standard optimizations to LLVM.

The StackTrek library consists of a set of macros that provide low-level facilities for stack switching, targeting the x86-64 architecture. Its design largely follows the formalization discussed in the previous sections. The macro library is useful by itself to a programmer who wants to write C programs using lexical effect handlers.

We first demonstrate the use of StackTrek by showing the compiled C code for the scheduler example in Figure 11. 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.

7.1 Overview

StackTrek provides a set of macros for stack switching: `HANDLE`, `RAISE`, and `RESUME`. Figure 11 demonstrates how they are used.

`HANDLE` (line 17) takes two arguments: the body of 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 body of code being

```c
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,
18     ({SINGLESHOT, yield},
19      {SINGLESHOT, fork}));
20 }
21 void scheduler(void (*f)(header_t*)){
22   spawn(f);
23   driver();
24 }
25 void job(header_t* P){
26   ···; RAISE(P, 0, ()); ···
27 }
28 void jobs(header_t* P, header_t* T){
29   for (int i = n_jobs; i > 0; i--){
30     RAISE(P, 1, (job));
31   }
32 }
33 int main(){
34   init_stack_pool();
35   scheduler(jobs);
36   destroy_stack_pool();
37 }
```

Figure 11. The scheduler example in C using StackTrek.

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.

RAISE (line 26 and 30) 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 formalized translation enforces at run time that a resumption can be resumed at most once. This restriction is similar to MultiCore OCaml [21], which supports single-shot resumptions but not multishot ones. As mentioned earlier, supporting multishot resumptions is a nongoal for us.

Nonetheless, LEXIC does allow multishot resumptions in restricted forms. When a resumption is resumed more than once, the resumption stack needs to be copied to preserve the original state of the resumption. A proper implementation should also copy the header frame, as it is part of the resumption's state. However, although copying a header frame is straightforward, finding all the pointers to the header frame and updating them is not. Thus, we have chosen to make all copies of a resumption share the same header frame to avoid searching for and updating pointers to the header frame. Because the header frame is shared, only one copy of the resumption can be active on the stack at a time. These choices help keep the implementation simple while enabling the programmer to use multishot resumptions in a controlled manner.

The assembly sequences that StackTrek uses for stack switching are written as naked C functions. As an example, the function in Figure 12 is responsible for the stack switch when raising an effect

```
1  __attribute__((naked, preserve_none))
2  int64_t save_switch_and_run_handler(intptr_t* env, int64_t arg, void* exec, void* func) {
3      __asm__ (
4          "movq 0(%%rdx), %%rax" // Load sp from the exchanger
5          "movq %%rsp, 0(%%rdx)" // Save sp to the exchanger
6          "movq %%rax, %%rsp" // Switch to the new stack
7          "jmpq *%%rcx" // Call the handler; the first three arguments are already in the right registers
8      );}
```

Figure 12. Stack-switching function for RAISE

with the RAISE macro. It is convenient that the assembly sequences are inside C functions, as the C compiler will respect the calling convention of the C functions and thus save and restore the register state (Section 7.2). We will call such functions *stack-switching functions.*

7.2 Calling Convention

In Salt, which is an abstract machine, the program state is stored on the stack, and the registers are used only to move 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. But when control is transferred to another function, the values of the registers need to be preserved across the function call. The System V x86-64 calling convention, 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. The OCaml compiler uses a different calling convention where all the registers are caller-saved. We will call this *preserve-none* calling convention.

The choice of the calling convention influences the amount of work that needs to be done during a stack switch. In MultiCore OCaml which uses preserve-none calling convention, stack switching is cheap, because there is no register state to save. However, preserve-none calling convention is not without cost. It may lead to higher overhead than the x86-64 calling convention. Consider a function A that uses one register **r** and calls function B twice. If using preserve-none, **r** needs to be saved and restored before and after each call to B even if B does not use **r**, costing 4 instructions, whereas in x86-64, by using a callee-saved register, **r** only needs to be saved and restored once in the prelogue and the epilogue of A, costing just 2 instructions.

In lexic, we use the x86-64 calling convention for most user functions, and we use preserve-none calling convention for the stack-switching functions and functions that implement effect operations. To indicate to the LLVM compiler that we want the preserve-none calling convention, we use the preserve_none attribute in the function declaration, as line 1 of Figure 12 shows. Using preserve-none calling convention significantly simplifies the implementation of StackTrek, because the stack-switching functions are no longer responsible for saving and restoring callee-saved registers, which is normally done by calling setjmp and longjmp. Using preserve-none calling convention also facilitates optimizations, as we will discuss soon. Furthermore, as we retain the x86-64 calling convention for most user functions, we mitigate the potential overhead associated with preserve-none calling convention.

Lifting the overhead out of the hot path. When an operation is raised in a hot path, the overhead of saving and restoring callee-saved registers can be significant. Consider line 30 in Figure 11, where a fork operation is raised in a tight loop. If the x86-64 calling convention is used, the callee-saved registers need to be saved inside the stack-switching function of RAISE every time fork is raised. This overhead will dominate the running time of the loop. In contrast, if preserve-none calling convention is used for the stack-switching functions, since the compiler knows that the call to these functions will potentially clobber all the registers, it includes every register in the use-set of the current function. Consequently, the compiler will save and restore these registers in the prelogue and the epilogue of the current function, incurring the overhead only once.

Exposing tail-call optimization opportunities. Using preserve-none calling convention also eliminates the need to restore callee-saved registers after a stack switch. As a result, performing a stack switch looks exactly like calling and returning from a function call, and the compiler can readily apply all the optimizations in its arsenal. For example, consider the call to `RESUME` in the tail position of `yield` at line 6. If the x86-64 calling convention is used, `RESUME` will need to restore the callee-saved registers after the call to a stack-switching function. In contrast, if preserve-none calling convention is used, no register restoration is needed, and the call to the stack-switching function becomes a tail call, so the compiler can deallocate the stack frame of `yield` before calling the stack-switching function. This optimization is significant, because otherwise the `yield` frames would accumulate on the stack and eventually cause a stack overflow.

7.3 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. Similar to prior work on implementing stacks and continuations [6, 21], we use a stack pool to recycle recently deallocated stacklets. The stack pool is initialized at the beginning of the program and is a continuous block of memory divided into fixed-size stacklets. A global bitmap is used to keep track of the availability of the stacklets.

In principle, an external system like a garbage collector or a reference counter is needed to reclaim unreachable stacklets. That being said, StackTrek will release stacklets to the pool when it is certain that the stacklet is no longer needed. This happens when the handled body finishes executing, and when an abortive handler is invoked and the control is transferred to an earlier stacklet. StackTrek also provides a special macro `RESUME_FINAL` for programmers to indicate that this is the last time a resumption is resumed. Consequently, no copy of the stacklet will be made, and the stacklet will be released to the pool after it finishes.

Header frames. In the formalized translation from Lexi to Salt, a header frame contains a pointer to the environemnt 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 or abortive handlers do not incur any heap allocation.

8 EVALUATION

Setup of experiments. We evaluate lexic against other implementations of lexical effect handlers, on a benchmark suit maintained by the community [1]. We also include two case studies, Scheduler and Interruptible Iterator, that represent more complicated, real-world uses of effect handlers. The experiments were conducted on a workstation with a 2.9GHz CPU.

We compare Lexi 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* [24], which are a form of lexical handlers. We also use as baselines two languages that support dynamically scoped handler: Koka and OCaml. For Effekt and Koka, we use the latest versions available at the time of writing: Effekt 0.2.2 and Koka 3.1.1. We do not use the latest version of OCaml, as it has dropped the experimental support for multishot resumptions. Instead, we use MultiCore OCaml 4.12.0, of which the limited support for multishot resumptions suffices for the benchmarks we consider.

Table 1. Benchmarking Lexi, Effekt, Koka, and OCaml. Cells with N/A mean that the benchmarks could not be implemented in the respective systems.

Benchmarks	Running time (ms)				
	Lexi	Effekt	Koka (named)	Koka (regular)	OCaml
Countdown	0.4	101.2	5715.5	3619.1	2779.1
Fibonacci Recursive	657.6	1657.1	1693.9	1665.9	1525.8
Product Early	105.5	362.5	2026.9	1985.9	131
Iterator	0.4	122.3	646.7	375.3	335.6
NQueens	421	146.5	2778.4	2562.7	905.8
Tree Explore	265.9	421.5	409.2	377	212.1
Triples	307.6	65.6	2357.4	2931.9	439.7
Resume Nontail	169.9	166.2	2087.3	2022.1	250.8
Parsing Dollars	403.5	202.3	4730.5	5554.6	2514.8
Handler Sieve	889.2	6393	3859	3937	3863
Scheduler	278.5	477.6×10^3	N/A	N/A	492.3
Interruptible Iterator	368	5599×10^3	1497	N/A	N/A

The Effekt compiler has multiple back ends. For most benchmarks, we use the MLton back end, as it produces the fastest code. However, the MLton back end cannot handle several benchmarks, so we resort to other back ends of Effekt in these cases. We use the Node.js back end for the Handler Sieve benchmark and the Chez Scheme back end for the Scheduler and Interruptible Iterator benchmarks. The MLton back end fails on these benchmarks, likely 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.

Results of experiments. Lexi is the fastest system on 7 out of the 12 benchmarks. On the 5 benchmarks that Lexi is not the fastest, it is the second fastest.

Lexi 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 lexic 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 lexic is able to generate code amenable to such optimizations is a testament to the quality of the generated code. In particular, because lexic specially treats tail-resumptive handlers and allocate the header frame on the same stack, no assembly is involved in the computation, allowing the LLVM compiler to perform optimizations without hindrance.

Lexi is not as efficient on benchmarks that use multishot resumptions, such as NQueens and Triples. On these benchmarks, Effekt is the fastest.

The Scheduler benchmark is discussed in Section 2. We could not implement Scheduler in Koka with either regular handlers or named handlers, due to a compiler-internal error. Implementing the Scheduler benchmark in Effekt required an unsafe type cast. Effekt does not scale for this benchmark, as Figure 2 shows. As discussed in Section 2, Effekt requires an increasing amount of lifting as the number of installed Exn handlers increases due to recursion, effectively incurring the run-time cost of dynamically walking the stack. Interestingly, OCaml scales well for this benchmark, as its exception handlers are implemented via a different mechanism than effect handlers, by passing the code address and the stack location of an exception handler to the raise site.

The Interruptible Iterator benchmark is adapted from earlier work on bidirectional effects [13, 28]. An implementation of this benchmark can be found in an appendix. It uses bidirectional effects 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. Similar to 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. Effekt therefore does not scale for this benchmark, requiring quadratically increasing running time. We were able to implement Interruptible Iterator in Koka with named handlers, though not with regular handlers. We observe linear scaling with named handlers in Koka, as they are implemented by passing *evidence* of a handler to the raise site. However, across the benchmarks, Koka in general leads to the highest running times, as also observed by Müller et al. [15].

9 RELATED WORK

Zhang et al. [27] 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. [27] compile tunneled exceptions 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 [26] show that the loss of local reasoning with dynamically scoped effect handlers can be understood as a loss of *parametricity* [22]. They prove, for a type-and-effect system supporting *effect polymorphism*, that local reasoning principles are restored by lexical effect handlers using a parametricity argument. Effect polymorphism, while syntactically heavier-weight than second-class values used by Zhang et al. [27], 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 ours, freshly generate labels for each installed handler at run time. This generativity is also seen with multi-prompt delimited control [9], SML exception names [14], and effect instances in the Eff language [2]. SML's exception handlers and Eff's effect handlers are dynamically scoped.

Brachthäuser et al. [5] 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. [27], 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 [15]. Λ_{cap} is further compiled to System F [20]. System Ξ is similar to Lexi, in that both are intermediate languages where handlers are passed explicitly. It is different in that System Ξ imposes the second-class restriction on functions, handlers, and continuations, which simplifies lift inference.

WasmFX is a proposal for adding effect handlers to WebAssembly [16]. 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 use a generative semantics.

Xie et al. [24] present a type-and-effect system for first-class named handlers in Koka. Koka is initially designed around dynamically scoped handlers [11], and Xie et al. [23, 25] formalize the compilation of dynamically scoped handlers to multi-prompt delimited control, passing so-called *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. Our implementation targets a lower-level language, where control transfer is in the form of `jmp` and `call` instructions.

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 lexically scoping effect handlers not only affords local reasoning principles, as shown by previous work, but also enables good performance, as suggested by our empirical evaluation. We hope that this work will encourage language designers and implementers to explore lexical scoping as a viable alternative to dynamically scoped effect handlers.

REFERENCES

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

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015).

[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).

[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).

[5] 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

[6] 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

[7] 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

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. *The Java Language Specification* (SE 20 ed.). Oracle America, Inc. https://docs.oracle.com/javase/specs/jls/se20/jls20.pdf

[9] 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)*.

[10] 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

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

[12] 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

[13] 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

[14] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.

[15] 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

[16] 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

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

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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

A OPERATIONAL SEMANTICS FOR LEXI

The operational semantics of LEXI follow below. Reduction is defined as a small-step evaluation relation $\longrightarrow$ on configurations $\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel t \rangle$, which consist of code memory $\mathfrak{H}$, heap memory H, evaluation context K, environment E, and term t.

For brevity we define $\hat{E}(v)$ as a helper function that looks up variables in an environment E but otherwise passes through values untransformed. Formally,

$$\hat{E}(v) = \begin{cases} E(x) & \text{when } v \text{ is a variable } x \\ c & \text{when } v \text{ is a constant } c \end{cases}$$

As code memory $\mathfrak{H}$ is immutable throughout the program except at initialization, we omit it from the configuration in the rules below.

(TERMINATE) Termination. $\langle \square \parallel E \parallel \texttt{halt}\ c \rangle$ terminates the evaluation with value c.

(INIT) Initialization. $\langle [] \parallel [] \parallel \square \parallel [] \parallel \texttt{letrec}\ \mathfrak{L}_1 \mapsto V_1, \cdots, \mathfrak{L}_n \mapsto V_n\ \texttt{in}\ t \rangle$ steps to $\langle [\mathfrak{L}_1 \mapsto V_1, \cdots, \mathfrak{L}_n \mapsto V_n, \mathfrak{L}_{\text{init}} \mapsto \lambda().t] \parallel [] \parallel \bullet \parallel [] \parallel t \rangle$. This initializes the code memory $\mathfrak{H}$ with the definitions of functions. The heap is set to empty. The evaluation context is set to be a hole. The local environment is set to empty. The term to be evaluated is set to t, the entry point of the program.

(ARITH) Arithmetic. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = v_1 + v_2\ \texttt{in}\ t \rangle$ steps to $\left\langle H \parallel K \parallel E[x \mapsto \hat{E}(v_1) + \hat{E}(v_2)] \parallel t \right\rangle$.

(VALUE) Value. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = v\ \texttt{in}\ t \rangle$ steps to $\left\langle H \parallel K \parallel E[x \rightarrow \hat{E}(v)] \parallel t \right\rangle$.

(NEW) New Reference. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{newref}\ \langle v_1, \cdots, v_n \rangle\ \texttt{in}\ t \rangle$ steps to $\left\langle H[L \mapsto \left\langle \hat{E}(v_1), \cdots, \hat{E}(v_n) \right\rangle] \parallel K \parallel E[x \mapsto L] \parallel t \right\rangle$, where L is a fresh location, initialized to an n-element tuple of values.

(GET) Get Reference. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = \pi_i\ (v)\ \texttt{in}\ t \rangle$ steps to $\langle H \parallel K \parallel E[x \mapsto c_i] \parallel t \rangle$, where $\hat{E}(v)$ is some location L in the memory such that $H(L) = \langle c_1, \cdots, c_n \rangle$.

(SET) Set Reference. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = v[i] \leftarrow v'\ \texttt{in}\ t \rangle$ steps to $\left\langle H[L \mapsto \left\langle c_1, \cdots, \hat{E}(v'), \cdots, c_n \right\rangle] \parallel K \parallel E[x \mapsto \hat{E}(v')] \parallel t \right\rangle$, where $\hat{E}(v) = L$ and $H(L) = \langle c_1, \cdots, c_n \rangle$, in effect replacing the ith element of the tuple at location L with $\hat{E}(v')$.

(HANDLE) Handler. $\left\langle H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{handle}\ \mathfrak{L}_{\text{body}}\ \texttt{with}\ A\ \mathfrak{L}_{\text{op}}\ \texttt{under}\ v_{\text{env}}\ \texttt{in}\ t \right\rangle$ steps to $\left\langle H \parallel K \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \cdot \#L_A^{\mathfrak{L}_{\text{op}}, L_{\text{env}}} \square \parallel [x_{\text{env}} \rightarrow L_{\text{env}}, x_{\text{hdl}} \rightarrow L] \parallel t' \right\rangle$, where L is a fresh location, $\mathfrak{H}(\mathfrak{L}_{\text{body}}) = \lambda(x_{\text{env}}, x_{\text{hdl}}).t'$, and $\hat{E}(v_{\text{env}}) = L_{\text{env}}$.

(LEAVE) Leave Handler. $\left\langle H \parallel K \cdot (E, \texttt{let}\ x' = \square\ \texttt{in}\ t) \cdot \#L_A^{\mathfrak{L}_{\text{op}}, L_{\text{env}}} \square \parallel E' \parallel v\ \texttt{end} \right\rangle$ steps to $\left\langle H \parallel K \parallel E[x' \mapsto \hat{E'}(v)] \parallel t \right\rangle$.

(APP) Application. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = v_1(\overline{v_2})\ \texttt{in}\ t \rangle$ steps to $\left\langle H \parallel K \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \parallel \left[\overline{x \mapsto \hat{E}(v_2)}\right] \parallel t' \right\rangle$, where $\hat{E}(v_1) = \mathfrak{L}$ and $\mathfrak{H}(\mathfrak{L}) = \lambda\overline{x}.t'$.

(RET) Return. $\langle H \parallel K \cdot (E, \texttt{let}\ x' = \square\ \texttt{in}\ t) \parallel E' \parallel v\ \texttt{end} \rangle$ steps to $\left\langle H \parallel K \parallel E[x' \mapsto \hat{E'}(v)] \parallel t \right\rangle$.

(RAISE) Raise Handler. $\left\langle H \parallel K \cdot \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{general}}\square\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise general}\ v_1 v_2\ \texttt{in}\ t\right\rangle$ steps to $\left\langle H[L_k \mapsto \texttt{cont}\ \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{general}}\square\right) \cdot K' \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t)] \parallel K \parallel \left[x_{\mathrm{env}} \to L_{\mathrm{env}}, y \to \hat{E}(v_2), k \to L_k\right] \parallel t'\right\rangle$, where L_k is fresh, $\hat{E}(v_1) = L$, and $\mathfrak{H}(\mathfrak{L}_{\mathrm{op}}) = \lambda(x_{\mathrm{env}}, y, k).t'$.

(RESUME) Resume Handler. $\langle H \parallel K \parallel E \parallel \texttt{let}\ x = \texttt{resume}\ v_1 v_2\ \texttt{in}\ t\rangle$ steps to $\left\langle H[L_k \mapsto \texttt{ns}] \parallel K \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \cdot K' \parallel E'[x' \mapsto \hat{E}(v_2)] \parallel t'\right\rangle$, where $\hat{E}(v_1) = L_k$, $H(L_k) = \texttt{cont}\ K' \cdot (E', \texttt{let}\ x' = \square\ \texttt{in}\ t')$.

(TAILRAISE) Tail-Resumptive Handler. $\left\langle H \parallel K \cdot \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{tail}}\square\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise tail}\ v_1 v_2\ \texttt{in}\ t\right\rangle$ steps to $\left\langle H \parallel K \cdot \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{tail}}\square\right) \cdot K' \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \parallel \left[x_{\mathrm{env}} \to L_{\mathrm{env}}, y \to \hat{E}(v_2)\right] \parallel t'\right\rangle$, where $\hat{E}(v_1) = L$ and $\mathfrak{H}(\mathfrak{L}_{\mathrm{op}}) = \lambda(x_{\mathrm{env}}, y).t'$.

(ABORTRAISE) Abortive Handler. $\left\langle H \parallel K \cdot \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{abort}}\square\right) \cdot K' \parallel E \parallel \texttt{let}\ x = \texttt{raise abort}\ v_1 v_2\ \texttt{in}\ t\right\rangle$ steps to $\left\langle H \parallel K \cdot \left(\#L^{\mathfrak{L}_{\mathrm{op}},L_{\mathrm{env}}}_{\texttt{abort}}\square\right) \cdot K' \cdot (E, \texttt{let}\ x = \square\ \texttt{in}\ t) \parallel \left[x_{\mathrm{env}} \to L_{\mathrm{env}}, y \to \hat{E}(v_2)\right] \parallel t'\right\rangle$, where $\hat{E}(v_1) = L$ and $\mathfrak{H}(\mathfrak{L}_{\mathrm{op}}) = \lambda(x_{\mathrm{env}}, y).t'$.

(EXIT) Exit the Program. $\langle K \parallel E \parallel \texttt{let}\ x = \texttt{exit}\ v\ \texttt{in}\ t\rangle$ steps to $\left\langle K \parallel E \parallel \texttt{halt}\ \hat{E}(v)\right\rangle$.

B OPERATIONAL SEMANTICS FOR SALT

The operational semantics of the trarget language SALT follow below. SALT is modelled as a register machine with separate code memory $\mathfrak{H}$ and heap memory H. Code memory $\mathfrak{H}$ maps locations to sequences of instructions that end in either a `jmp` or a `return`. We define a helper function $\hat{\mathfrak{H}}(\mathfrak{L}^i)$ that gives the instruction from a code memory location $\mathfrak{L}^i$.

$$\hat{\mathfrak{H}}(\mathfrak{L}^i) = \begin{cases} \iota_i & \text{when } \mathfrak{H}(\mathfrak{L}) = \iota_0; \iota_1; \iota_2; \cdots \\ \texttt{return} & \text{when } \mathfrak{H}(\mathfrak{L}) = \iota_0; \cdots; \iota_{i-1}; \texttt{return} \\ \texttt{jmp}\ o & \text{when } \mathfrak{H}(\mathfrak{L}) = \iota_0; \cdots; \iota_{i-1}; \texttt{jmp}\ o \end{cases}$$

For brevity, we define $\ell^i = \overbrace{\texttt{next} \cdots \texttt{next}}^{i}(\ell)$, which denotes the ith successor of location ℓ. We additionally define $\hat{R}(o)$ as shorthand for reading a register or a constant, formally

$$\hat{R}(o) = \begin{cases} R(\mathbf{r}) & \text{when } o = \mathbf{r} \\ o & \text{otherwise} \end{cases}$$

$\hat{H}(\ell)$ reads from heap memory. Formally, given a memory location L^i,

$$\hat{H}(L^i) = \begin{cases} v_i & \text{when } H(L) = \langle v_0, v_1, v_2, \cdots\rangle \\ v_i & \text{when } H(L) = \texttt{nil} :: v_1 :: v_2 :: v_3 :: \cdots \end{cases}$$

The instructions of SALT follow:

ADD – arithmetic.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{add}\ \mathbf{r}, o}{\langle H \parallel R\rangle \longrightarrow \left\langle H \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r} \mapsto R(\mathbf{r}) + \hat{R}(o)]\right\rangle}$$

MKSTK – stack construction.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{mkstk}\ \mathbf{r} \quad L \text{ is fresh}}{\langle H \parallel R\rangle \longrightarrow \langle H[L \mapsto \texttt{nil}] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r} \mapsto L]\rangle}$$

SALLOC – stack allocation.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{salloc}\ n \quad R(\mathbf{sp}) = L_{\text{sp}}^{j} \quad H(L_{\text{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_j}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L_{\text{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_j :: \overbrace{\textsf{ns} :: \cdots :: \textsf{ns}}^{n}] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{sp} \mapsto L_{\text{sp}}^{j+n}]\right\rangle}$$

PUSH – stack push.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{push}\ o \quad R(\mathbf{sp}) = L_{\text{sp}}^{j} \quad H(L_{\text{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_j \quad L_{\text{sp}}^{j+1} \text{ is fresh}}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L_{\text{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_j :: \hat{R}(o)] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{sp} \mapsto L_{\text{sp}}^{j+1}]\right\rangle}$$

SFREE – stack deallocation.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{sfree}\ n \quad R(\mathbf{r}) = L_{\text{sp}}^{j} \quad H(L_{\text{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_j \quad j \geq n}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L_{\text{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_{j-n}] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{sp} \mapsto L_{\text{sp}}^{j-n}]\right\rangle}$$

POP – stack pop.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{pop}\ \mathbf{r} \quad R(\mathbf{sp}) = L_{\text{sp}}^{j} \quad H(L_{\text{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_j \quad j > 0}{\langle H \parallel R\rangle \longrightarrow \left\langle HL_{\text{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_{j-1} \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r} \mapsto v_j, \mathbf{sp} \mapsto L_{\text{sp}}^{j-1}]\right\rangle}$$

MALLOC – heap allocation.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{malloc}\ \mathbf{r}_d, i \quad L \text{ is fresh}}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L \mapsto \left\langle \overbrace{\textsf{ns}, \cdots, \textsf{ns}}^{i} \right\rangle] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r} \mapsto L]\right\rangle}$$

MOV – register assignment.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{mov}\ \mathbf{r}_d, o}{\langle H \parallel R\rangle \longrightarrow \left\langle H \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r}_d \mapsto \hat{R}(o)]\right\rangle}$$

MOV (SP) – stack pointer assignment. Writes to the stack pointer register **sp** are special in that they truncate the stack in memory as well.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{mov}\ \mathbf{sp}, o \quad \hat{R}(o) = L^{j} \quad H(L) = \texttt{nil} :: v_1 :: \cdots :: v_k \quad j \leq k}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L \mapsto \texttt{nil} :: v_1 :: \cdots :: v_j] \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{sp} \mapsto L^{j}]\right\rangle}$$

LOAD – memory load.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{load}\ \mathbf{r}_d, [\mathbf{r}_s + j] \quad R(\mathbf{r}_s) = L}{\langle H \parallel R\rangle \longrightarrow \left\langle H \parallel R[\mathbf{ip} \mapsto \ell^{+1}, \mathbf{r}_d \mapsto \hat{H}(L^{j})]\right\rangle}$$

STORE – memory store.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{store}\ [\mathbf{r}_d + j], o_s \quad R(\mathbf{r}_d) = L \quad H(L) = \langle v_0, \cdots, v_{n-1}\rangle \quad j < n}{\langle H \parallel R\rangle \longrightarrow \left\langle H[L \mapsto \left\langle v_0, \cdots, v_{j-1}, \hat{R}(o_s), \cdots, v_{n-1}\right\rangle] \parallel R[\mathbf{ip} \mapsto \ell^{+1}]\right\rangle}$$

CALL – function call.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{call}\ o \quad R(\mathbf{sp}) = L^{j}_{\mathrm{sp}} \quad H(L_{\mathrm{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_j \quad \hat{R}(o) = \ell_{\mathrm{dest}}}{\langle H \parallel R \rangle \longrightarrow \left\langle H[L_{\mathrm{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_j :: \ell^{+1}] \parallel R[\mathbf{ip} \mapsto \ell_{\mathrm{dest}}] \right\rangle}$$

JMP – unconditional jump.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{jmp}\ o \quad \hat{R}(o) = \ell_{\mathrm{dest}}}{\langle H \parallel R \rangle \longrightarrow \langle H \parallel R[\mathbf{ip} \mapsto \ell_{\mathrm{dest}}] \rangle}$$

RETURN – function return.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{return} \quad R(\mathbf{sp}) = L^{j}_{\mathrm{sp}} \quad H(L_{\mathrm{sp}}) = \texttt{nil} :: v_1 :: \cdots :: v_{j-1} :: \ell_{\mathrm{dest}}}{\langle H \parallel R \rangle \longrightarrow \left\langle H[L_{\mathrm{sp}} \mapsto \texttt{nil} :: v_1 :: \cdots :: v_{j-1}] \parallel R[\mathbf{ip} \mapsto \ell_{\mathrm{dest}}] \right\rangle}$$

HALT – program termination.

$$\frac{R(\mathbf{ip}) = \ell \quad \hat{\mathfrak{H}}(\ell) = \texttt{halt}}{\langle H \parallel R \rangle !}$$

C TRANSLATION

Translation from LEXI to SALT is defined by the following rules. The translation of a program P is denoted $[\![P]\!]$, with forms for translating values and statements.

Translating Handler Annotations. Translating a handler annotation simply moves the annotation to the desired register.

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

Translating Values. Translating values simply moves the value to the desired register, and requires knowledge of the current variable environment Γ. Variables are read at positive offsets from the stack pointer.

$[\![x_i]\!]^{\mathbf{r}}_{(x_{n-1},\cdots,x_0)} = \texttt{load}\ \mathbf{r}, [\mathbf{sp} + i]$
$[\![i]\!]^{\mathbf{r}}_{\Gamma} = \texttt{mov}\ \mathbf{r}, i$
$[\![\mathfrak{L}]\!]^{\mathbf{r}}_{\Gamma} = \texttt{mov}\ \mathbf{r}, \mathfrak{L}$

Translating Functions and Programs. Translating a function $\lambda\overline{x}.t$ requires the translation of the body t with the variables $\overline{x}$ in scope. This requires knowledge of the number of nested let-bindings in t.

$$\text{Define NUM\_LET}(t) = \begin{cases} 0 & t = x \\ 1 + \text{NUM\_LET}(t') & t = \texttt{let}\ x = e\ \texttt{in}\ t' \end{cases}$$

With that defined, the translation $[\![\lambda\overline{x}.t]\!]$ for a function $\lambda\overline{x}.t$ is as follows:

$$[\![\lambda\overline{x}.t]\!] = \begin{array}{l} \texttt{push}\ \mathbf{rn} \\ \cdots \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{(x_n,\cdots x_1)} \\ \texttt{sfree}\ \mathbf{n} + \text{NUM\_LET}(\mathbf{t}) \\ \texttt{return} \end{array}$$

Translating a whole program – collection of functions – simply maps to translating each function in the program.

$[\![\{L_1 \mapsto \lambda\overline{x}.t_1, \cdots, L_n \mapsto \lambda\overline{x}.t_n\}]\!] = \{L_1 \mapsto [\![\lambda\overline{x}.t_1]\!], \cdots, L_n \mapsto [\![\lambda\overline{x}.t_n]\!]\}$

Translating Statements. As LEXI is in ANF, the translation of a statement is straightforward, as each statement is a let-bound expression.

$[\![v\ \texttt{end}]\!]_\Gamma$. The translation of the term's base case, which is a value, is the translation of the value: $[\![v]\!]_\Gamma^{\mathbf{r1}}$

$[\![\texttt{let}\ x = v\ \texttt{in}\ t]\!]_\Gamma$. The translation of a let-bound value is the translation of the value v followed by the translation of the body t with the variable x in scope, pushed on to the stack appropriately.

$$\begin{array}{l} [\![v]\!]_\Gamma^{\mathbf{r1}} \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = v_1\ \texttt{arith}\ v_2\ \texttt{in}\ t]\!]_\Gamma$. The translation of an let-bound arithmetic operation is the translation of the two values v_1 and v_2 followed by the arithmetic operation, pushing the result on to the stack, and then translating the body t with the variable x in scope.

$$\begin{array}{l} [\![v_1]\!]_\Gamma^{\mathbf{r2}} \\ [\![v_2]\!]_\Gamma^{\mathbf{r1}} \\ \texttt{arith}\ \mathbf{r2}, \mathbf{r1} \\ \texttt{push}\ \mathbf{r2} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = v(v_1, \cdots, v_n)\ \texttt{in}\ t]\!]_\Gamma$. Let-bound function calls evaluate the arguments in reverse order, storing their results in the register file, then call the function, pushing the result on to the stack, and then translate the body t with the variable x in scope.

$$\begin{array}{l} [\![v]\!]_\Gamma^{\mathbf{r0}} \\ [\![v_n]\!]_\Gamma^{\mathbf{rn}} \\ \ldots \\ [\![v_1]\!]_\Gamma^{\mathbf{r1}} \\ \texttt{call}\ \mathbf{r0} \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = \texttt{newref}\ \langle v_1, \cdots, v_n\rangle\ \texttt{in}\ t]\!]_\Gamma$. Let-bound allocations first allocate a new reference, then evaluate each value in the vector in order, store the results on the vector, and finally translate the body t with the variable x in scope.

$$\begin{array}{l} \texttt{malloc}\ \mathbf{r2}, i \\ [\![v_1]\!]_\Gamma^{\mathbf{r1}} \\ \texttt{store}\ [\mathbf{r2} + 0], \mathbf{r1} \\ \ldots \\ [\![v_n]\!]_\Gamma^{\mathbf{r1}} \\ \texttt{store}\ [\mathbf{r2} + n - 1], \mathbf{r1} \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = \pi_i(v)\ \texttt{in}\ t]\!]_\Gamma$. Let-bound get operations first evaluate the value v, then load the ith element of the vector at the location v, push the result on to the stack, and finally translate the body t with the variable x in scope.

$$\begin{array}{l} \llbracket v \rrbracket_{\Gamma}^{\mathbf{r1}} \\ \texttt{load}\ \mathbf{r1}, [\mathbf{r1} + i] \\ \texttt{push}\ \mathbf{r1} \\ \llbracket t \rrbracket_{\Gamma,x} \end{array}$$

$\llbracket \texttt{let}\ x = v_1[i] \leftarrow v_2\ \texttt{in}\ t \rrbracket_{\Gamma}$. Let-bound set operations first evaluate the values v_1 and v_2, then load the vector at the location v_1, store the value v_2 at the ith element of the vector, push the result on to the stack, and finally translate the body t with the variable x in scope.

$$\begin{array}{l} \llbracket v_2 \rrbracket_{\Gamma}^{\mathbf{r2}} \\ \llbracket v_1 \rrbracket_{\Gamma}^{\mathbf{r1}} \\ \texttt{store}\ [\mathbf{r1} + i], \mathbf{r2} \\ \texttt{push}\ \mathbf{r2} \\ \llbracket t \rrbracket_{\Gamma,x} \end{array}$$

$\llbracket \texttt{let}\ x = \texttt{handle}\ \mathfrak{L}_{\mathbf{body}}\ \texttt{with}\ A\ \mathfrak{L}_{\mathbf{op}}\ \texttt{under}\ v_{\mathbf{env}}\ \texttt{in}\ t \rrbracket_{\Gamma}$. Let-bound handler expressions move the annotation, environment, operation function and body function to the desired registers, then invoke the stack switching function $\mathfrak{L}_{\text{handle}}$. If the annotation indicates a tail-resumptive or abortive handler, $\mathfrak{L}_{\text{handle}}$ is replaced with $\mathfrak{L}_{\text{handle\_same\_stack}}$. Next, the result is pushed on to the stack, and finally the body t is translated with the variable x in scope. $\mathfrak{L}_{\text{handle}}$ and $\mathfrak{L}_{\text{handle\_same\_stack}}$ are builtin functions that will be defined later.

$$\begin{array}{l} \llbracket A \rrbracket^{\mathbf{r4}} \\ \llbracket v_{\text{env}} \rrbracket_{\Gamma}^{\mathbf{r3}} \\ \texttt{mov}\ \mathbf{r2}, \mathfrak{L}_{\text{op}} \\ \texttt{mov}\ \mathbf{r1}, \mathfrak{L}_{\text{body}} \\ \texttt{call}\ \mathfrak{L}_{\text{handle}} // \text{or}\ \mathfrak{L}_{\text{handle\_same\_stack}} \\ \texttt{push}\ \mathbf{r1} \\ \llbracket t \rrbracket_{\Gamma,x} \end{array}$$

$\llbracket \texttt{let}\ x = \texttt{raise general}\ v_1 v_2\ \texttt{in}\ t \rrbracket_{\Gamma}$. Let-bound general raise expressions first evaluate the values v_1 and v_2, then invoke a builtin function $\mathfrak{L}_{\text{raise}}$ for raising the exception. $\mathfrak{L}_{\text{raise}}$ will be defined later.

$$\begin{array}{l} \llbracket v_2 \rrbracket_{\Gamma}^{\mathbf{r2}} \\ \llbracket v_1 \rrbracket_{\Gamma}^{\mathbf{r1}} \\ \texttt{call}\ \mathfrak{L}_{\text{raise}} \\ \texttt{push}\ \mathbf{r1} \\ \llbracket t \rrbracket_{\Gamma,x} \end{array}$$

$\llbracket \texttt{let}\ x = \texttt{raise tail}\ v_1 v_2\ \texttt{in}\ t \rrbracket_{\Gamma}$. When raising a tail-resumptive handler, the the operation function is loaded into register **r3** from the pointer to the header frame, and the environment is also loaded from the header frame into register **r1**. The control is then transferred to the operation function.

$$\begin{array}{l} [\![v_2]\!]^{\mathbf{r2}}_{\Gamma} \\ [\![v_1]\!]^{\mathbf{r1}}_{\Gamma} \\ \texttt{load}\ \mathbf{r3}, [\mathbf{r1}+3] \\ \texttt{load}\ \mathbf{r1}, [\mathbf{r1}+2] \\ \texttt{call}\ \mathbf{r3} \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = \texttt{raise abort}\ v_1 v_2\ \texttt{in}\ t]\!]_{\Gamma}$. Similar to tail-resumptive handlers, but the abortive handler additionally cut the stack to the header frame, and then deallocate 4 bytes of the header frame, exposing the previous frame on the stack. The control is then jumped to the operation function. Notice that `jmp` is used instead of `return`. This is because the return address of the previous frame is already on the stack.

$$\begin{array}{l} [\![v_2]\!]^{\mathbf{r2}}_{\Gamma} \\ [\![v_1]\!]^{\mathbf{r1}}_{\Gamma} \\ \texttt{load}\ \mathbf{r3}, [\mathbf{r1}+3] \\ \texttt{mov}\ \mathbf{sp}, \mathbf{r1} \\ \texttt{load}\ \mathbf{r1}, [\mathbf{r1}+2] \\ \texttt{sfree}\ 4 \\ \texttt{jmp}\ \mathbf{r3} \end{array}$$

$[\![\texttt{let}\ x = \texttt{resume}\ v_1\ v_2\ \texttt{in}\ t]\!]_{\Gamma}$. Resumptions simply call the $\mathfrak{L}_{\text{resume}}$ builtin function, passing the pointer of the resumption and the resumed value.

$$\begin{array}{l} [\![v_2]\!]^{\mathbf{r2}}_{\Gamma} \\ [\![v_1]\!]^{\mathbf{r1}}_{\Gamma} \\ \texttt{call}\ \mathfrak{L}_{\text{resume}} \\ \texttt{push}\ \mathbf{r1} \\ [\![t]\!]_{\Gamma,x} \end{array}$$

$[\![\texttt{let}\ x = \texttt{exit}\ v\ \texttt{in}\ t]\!]_{\Gamma}$. Exit moves the value v to register **r1**, then terminate at the next `halt` instruction.

$$\begin{array}{l} [\![v]\!]^{\mathbf{r1}}_{\Gamma} \\ \texttt{halt} \end{array}$$

In addition, the following builtin functions are installed in the generated SALT code for constructing, invoking, and resuming handlers.

Handler Creation. Handler creation expects four arguments: In **r1**, the location of the handler body, in **r2**, the location of the handler operation, in register **r3**, the location of the environment, and finally, in register **r4**, the annotation of the handler.

$\mathfrak{L}_{\text{handle}}$:

```
mov r5, sp
mkstk sp
// sets up the header frame:// the operation, environment, anotation and exchanger
push r2; push r3; push r4; push r5
mov r6, r1
mov r2, sp
mov r1, r3
// call body, passing the environment in r1 and the header of the stack in r2
call r6
pop r2
sfree 3
mov sp, r2
return
```

Tail-Resumtive and Abortive Handler Creation. We can generate simpler code for tail-resumptive or arbortive handlers, as no stack needs to be allocated. As before, the handler body is expected in register 1, the handler operation in register 2, the environment in register 3, and the handler type in register 4.

$\mathfrak{L}_{\text{handle\_same\_stack}}$:

```
// sets up the header frame:// the operation, environment, anotation and a ns value for exchanger
// type, and a blank for the stack exchanger
push r2; push r3; push r4; push ns
mov r6, r1
mov r2, sp
mov r1, r3
// call body, passing the environment in r1 and the pointer to the header frame in r2
call r6
sfree 4
return
```

Raising a Handler. To raise a handler, we expect the pointer to the stack header in register 1 and the argument to the handler in register 2.

$\mathfrak{L}_{\text{raise}}$:

```
load r4, [r1 + 0]
store [r1 + 0], sp
mov sp, r4
load r5, [r1 + 3]
malloc r3, 1
store [r3 + 0], r1
load r1, [r1 + 2]
// invoke the handler, passing in r1 the environment,
// r2 the argument, and r3 the resumption
call r5
return
```

Resuming from a Handler. To resume from a handler, we expect the pointer to the resumption object in register 1 and the argument to the resumption in register 2.

$$\begin{array}{l}
\mathfrak{L}_{\text{resume}} : \\
\texttt{load}\ \mathbf{r3}, [\mathbf{r1} + 0] \\
\texttt{store}\ [\mathbf{r1} + 0], \texttt{ns} \\
\texttt{load}\ \mathbf{r4}, [\mathbf{r3} + 0] \\
\texttt{store}\ [\mathbf{r3} + 0], \mathbf{sp} \\
\texttt{mov}\ \mathbf{r1}, \mathbf{r2} \\
\texttt{mov}\ \mathbf{sp}, \mathbf{r4} \\
\texttt{return}
\end{array}$$

D SIMULATION PROOF

To setup our simulation proof, we have to relate the runtime configuration of a Lexi program to that of a Salt program.

Relating Values $v \overset{\textbf{value}}{\sim} v'$. Basic values in Lexi and Salt consist of just numbers, dummies, and locations (into instruction memory and heap memory). These values we can relate directly. We assume that the labels in Lexi and the locations in Salt use the same set of labels, and they are related by equality.

- $i \underset{\Xi}{\overset{\text{value}}{\sim}} i$
- $\texttt{ns} \underset{\Xi}{\overset{\text{value}}{\sim}} \texttt{ns}$
- $\mathfrak{L} \underset{\Xi}{\overset{\text{value}}{\sim}} \mathfrak{L}$
- $L \underset{\Xi}{\overset{\text{value}}{\sim}} L$

In the text that follows, we use the term `stack` *and* `stack segment` *interchangeably. They both mean the stack in Salt.*

Relating Local Environments to Stack Segments $E \overset{\textbf{local env}}{\sim} s$. The relation is constructed by the inference rule below, which is self-explanatory.

$$\frac{v_i \overset{\text{value}}{\sim} v'_i}{[x_1 \longrightarrow v_1, \cdots, x_n \longrightarrow v_n] \overset{\text{local env}}{\sim} \texttt{nil} :: v'_1 :: \cdots v'_n}$$

In the text that follows, we use the the metavariable F to denote an activation frame $(E, \texttt{let}\ x = \square\ \texttt{in}\ t)$ in Lexi. We use the metavariable $\mathbb{F}$ to denote a sequence of activation frames $F_1 \cdot \cdots \cdot F_n$ in Lexi.

Relating Term to Instruction Sequence $F \overset{\textbf{ins}}{\sim} I$. This rule is used in two places: (1) to relate an activation frame in Lexi to a stack segment in Salt, and (2) to relate the current environment and term in Lexi to the current instruction sequence in Salt.

The instruction sequence corresponding to a a term in Lexi is the sequence of corresponding translated instructions, plus the instructions to free the stack segment allocated for the frame.

$(E, t) \overset{\text{ins}}{\sim} [\![t]\!]_{\mathsf{dom}(E)}\ ; \texttt{sfree}\ \ (|E| + \text{NUM\_LET}(t)); \texttt{return}$

One exception to the above rule is when the term is a `halt`. In this case, the instruction sequence is simply a `halt`.

$(E, \texttt{halt}\ c) \overset{\text{ins}}{\sim} \texttt{halt}$

Relating Activation Frame to Stacks $F \overset{\textbf{frame}}{\sim} s$. Using the previous two relations, we can construct the relation between an activation frame in Lexi and a stack segment in Salt that corresponds to the frame. The stack segment consists of the stack for the local environment and a return address. The return address is a memory location that starts the instruction sequence corresponding to the term in the frame. More formally:

$$\frac{(E, t) \overset{\text{ins}}{\sim} I \quad \mathfrak{H}(\mathfrak{L}) = \iota_0 :: \cdots \iota_{i-1} :: I \quad E \overset{\text{local env}}{\sim} s}{(E, \mathtt{let}\ x = \square\ \mathtt{in}\ t) \overset{\text{frame}}{\sim} s :: \mathfrak{L}^i}$$

Relating Sequence of Activation Frames to Stacks $\mathbb{F} \overset{\textbf{frames}}{\sim} s$. A sequence of activation frames in Lexi simply corresponds the concatenation of their corresponding stack segments in Salt. Here, we use @ to denote the concatenation of two stack segments.

$$\frac{F_i \overset{\text{frames}}{\sim} s_i}{F_1 \cdot \cdots \cdot F_n \overset{\text{frames}}{\sim} \mathtt{nil}@s_1@\cdots@s_n}$$

In the text that follows, we use the the metavariable $\mathbb{K}$ to denote an evaluation context consisting of a leading handler frame followed by a sequence of activation frames. We call such evaluation context handler-led context. A sequence of handler-led contexts is denoted by $\overline{\mathbb{K}}$.

Relating Sequence of Handler-Led Contexts to Stacks $\overset{\downarrow}{\overline{\mathbb{K}}} \overset{\textbf{hdl-led ctx}}{\sim} \left(\overset{\downarrow}{L_{\textbf{in}}}, \overset{\downarrow}{s_{\textbf{in}}}, \overset{\uparrow}{H_{\textbf{stack}}}, \overset{\uparrow}{L_{\textbf{out}}}, \overset{\uparrow}{s_{\textbf{out}}}\right)$

. The relation between a sequence of handler-led contexts and a stack is constructed recursively. It is complicated for two reasons: (1) the stack layout of each handler-led context depends on the type of the handler, so there are three rules, one for each type of handler, and (2) for a general handler whose header frame is on a new stack, the exchanger on the header frame points to the top of the previous stack. Therefore, the stack needs to be constructed one by one, with the inputs from the previous stack, and the outputs to the next stack. For this purpose, the relation takes in L_{in}, the bottom of the previous stack, and s_{in}, the previous stack; the two can be used to calculate the location of the top of the previous stack. It also outputs L_{out} and s_{out} to be used for constructing the next stack. The inference rules also populates the heap H_{stack} with the stacks. Now we present the inference rules one by one.

The base case is when the sequence of handler-led contexts is a hole. In this case, inputs and outputs are the same, and no stacks is added to the heap.

$$\bullet \overset{\text{hdl-led ctx}}{\sim} (L, s, \{\}, L, s)$$

When encountering a general handler, a new stack s' is constructed. The location of the stack base L is chosen to be the same as the label of the handler L. The annotation on the header frame is set to 0, and the exchanger is set to the top of the previous stack. A recursive call is made to construct the rest of the stacks, with the inputs being L and s'. Finally, the outputs of the relations are the ones from the recursive call, with the previous stack $L_{\text{in}} \mapsto s_{\text{in}}$ added H_{stack}.

$$\frac{\mathbb{F} \overset{\text{frames}}{\sim} s \quad s' = \mathrm{nil} :: \mathfrak{L}_{\text{op}} :: L_{\text{env}} :: 0 :: L_{\text{in}}^{|s_{\text{in}}|}@s \quad \overline{\mathbb{K}} \overset{\text{hdl-led ctx}}{\sim} (L, s', H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}{\left(\left(\#L_{\mathtt{general}}^{\mathfrak{L}_{\text{op}}, L_{\text{env}}} \square\right) \cdot \mathbb{F} \cdot \overline{\mathbb{K}}\right) \overset{\text{hdl-led ctx}}{\sim} (L_{\text{in}}, s_{\text{in}}, \{L_{\text{in}} \mapsto s_{\text{in}}\} \cup H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}$$

For tail-resumptive and abortive handlers, the construction of the stack simply extend the input stack with the header frame of the handler. The only difference between the two is the annotation on the header frame.

$$\frac{\mathbb{F} \overset{\text{frames}}{\sim} s \quad s' = s_{\text{in}} :: \mathfrak{L}_{\text{op}} :: L_{\text{env}} :: 1 :: \mathsf{ns} :: @s \quad \overline{\mathbb{K}} \overset{\text{hdl-led ctx}}{\sim} (L_{\text{in}}, s', H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}{\left(\left(\#L_{\mathtt{tail}}^{\mathfrak{L}_{\text{op}}, L_{\text{env}}} \square\right) \cdot \mathbb{F} \cdot \overline{\mathbb{K}}\right) \overset{\text{hdl-led ctx}}{\sim} (L_{\text{in}}, s_{\text{in}}, H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}$$

$$\frac{\mathbb{F} \overset{\text{frames}}{\sim} s \quad s' = s_{\text{in}} :: \mathfrak{L}_{\text{op}} :: L_{\text{env}} :: 2 :: \mathtt{ns} :: @s \quad \overline{\mathbb{K}} \overset{\text{hdl-led ctx}}{\sim} (L_{\text{in}}, s', H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}{\left(\left(\#L^{\mathfrak{L}_{\text{op}}, L_{\text{env}}}_{\mathtt{abort}} \square\right) \cdot \mathbb{F} \cdot \overline{\mathbb{K}}\right) \overset{\text{hdl-led ctx}}{\sim} (L_{\text{in}}, s_{\text{in}}, H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}$$

Relating Evaluation Contexts and Environments to Stacks and sp $(K, E) \overset{\textbf{context}}{\sim} (H_{\textbf{stack}}, \ell_{\textbf{last}})$. The evaluation context in Lexi can be thought of as consisting of a sequence of activation frames $\mathbb{F}$ followed by one or more handler-led contexts $\overline{\mathbb{K}}$. We use the relations established earlier to construct stacks using these two components. The relation also takes in E, the local environment, as it is allocated on the last stack. We use a helper function `val` to extract the values from the local environment. We also abuse the notation of $[\![\cdot]\!]$ to be used on a list of values.

The relation is presented below. It extends the heap H_{stack} with the last stack, and also outputs the top location of the last stack, which is deemed as the current stack pointer. Also notice that we use L_0 as a distinguished base location for the first stack.

$$\frac{\mathbb{F} \overset{\text{frames}}{\sim} s \quad \overline{\mathbb{K}} \overset{\text{hdl-led ctx}}{\sim} (L_0, s, H_{\text{stack}}, L_{\text{out}}, s_{\text{out}}) \quad s' = s_{\text{out}} @ [\![\mathtt{val}(E)]\!]}{\left(\square \cdot \mathbb{F} \cdot \overline{\mathbb{K}}, E\right) \overset{\text{context}}{\sim} \left(H_{\text{stack}} \cup \left\{L^{|s'|}_{\text{out}} \mapsto s'\right\}, L^{|s'|}_{\text{out}}\right)}$$

Relating Data Heap $H \overset{\textbf{data heap}}{\sim} H$. The data heap in Lexi consists of tuples and resumptions. The rule below assumes without loss of generality that the data heap in Lexi consists of a single tuple and a single resumption. The relation for tuple is straightforward. The relation for resumption is a bit more involved. Like how we relate the evaluation context, the evaluation context part of the resumption is split into $\mathbb{F}$ and $\overline{\mathbb{K}}$, and the relation established earlier is used to construct the stacks. The output H_{stack} contains the stacks constructed for the resumption, but we need to discard the first stack from the collection. The reason is that the header frame of the first stack L should have its exchanger pointing to the top of the last frame, so we need to build it separately. Similar to the evaluation context relation, we also need to join the last stack L_{out} into the heap. Finally, the label of the resumption L_{rsp} points to a single-element tuple, which contains the location of the header frame of the first stack L^4.

$$\frac{\overline{v} \overset{\text{value}}{\sim} \overline{v'} \quad \mathbb{F} \overset{\text{frames}}{\sim} s \quad \overline{\mathbb{K}} \overset{\text{hdl-led ctx}}{\sim} (L, s, H_{\text{stack}}, L_{\text{out}}, s_{\text{out}})}{\begin{array}{l}\left\{L_{\text{tup}} \mapsto \langle \overline{v} \rangle, L_{\text{rsp}} \mapsto \mathtt{cont}\ \left(\#L^{\mathfrak{L}_{\text{op}}, L_{\text{env}}}_{\text{general}} \square\right) \cdot \mathbb{F} \cdot \overline{\mathbb{K}}\right\} \overset{\text{data heap}}{\sim} \\ \left\{L_{\text{tup}} \mapsto \left\langle \overline{v'} \right\rangle, L_{\text{rsp}} \mapsto \left\langle L^4 \right\rangle, L \mapsto \mathtt{nil} :: \mathfrak{L}_{\text{op}} :: L_{\text{env}} :: 0 :: L^{|s_{\text{out}}|}_{\text{out}} :: s, L_{\text{out}} \mapsto s_{\text{out}}\right\} \cup (H_{\text{stack}} \setminus L)\end{array}}$$

Relating Lexi code to Salt code $\mathfrak{H} \overset{\textbf{code memory}}{\sim} \mathfrak{H}$. Lexi code can be directly related to its compiled Salt equivalent.

$$\left\{\overline{\mathfrak{L} \mapsto \lambda \overline{x}.t}\right\} \overset{\text{code memory}}{\sim} \left\{\overline{\mathfrak{L} \mapsto [\![\lambda \overline{x}.t]\!]}\right\}$$

Proof of Theorem 2. For convenience, we copy the definition of the initial and final predicate below.

- $\text{initial}_{\text{Lexi}} \left(\mathtt{letrec}\ \overline{\mathfrak{L}_i \mapsto \lambda \overline{x}.t_i}\ \mathtt{in}\ \mathfrak{L}_{\text{main}}(), \left\langle \left\{\overline{\mathfrak{L}_i \mapsto \lambda \overline{x}.t_i}\right\} \parallel \{\} \parallel \square \parallel [] \parallel \mathtt{let}\ x = \mathfrak{L}_{\text{main}}(),\ \_ = \mathtt{exit}\ x\ \mathtt{in\ ns\ end}\right\rangle\right)$
- $\text{initial}_{\text{Salt}} \left(\left\{\overline{\mathfrak{L}_i \mapsto I_i}\right\}, \left\langle \left\{\overline{\mathfrak{L}_i \mapsto I_i},\ \mathfrak{L}_{\text{init}} \mapsto \mathtt{call}\ \mathfrak{L}_{\text{main}}; \mathtt{halt}\right\} \parallel \{L_{\text{stack}} \mapsto \mathtt{nil}\} \parallel \left\{\mathbf{sp} \mapsto L^0_{\text{stack}}, \mathbf{ip} \mapsto \mathfrak{L}^0_{\text{init}}\right\}\right\rangle\right)$
- $\text{final}_{\text{Lexi}} \left(\langle \mathfrak{H} \parallel H \parallel K \parallel E \parallel \mathtt{halt}\ i \rangle, i\right)$
- $\text{final}_{\text{Salt}} \left(\langle \mathfrak{H} \parallel H \parallel R[\mathbf{ip} \mapsto \ell, \mathbf{r1} \mapsto i] \rangle, i\right)$ where $\hat{\mathfrak{H}}(\ell) = \mathtt{halt}$

We first prove the first condition, that if $P_2 = [\![P_1]\!]$, then the initial configurations of the two programs are related, i.e., $M_1 \sim M_2$.

To establish relation $M_1 \sim M_2$, we need to show that its premises hold. The first premise holds: the term in Lexi makes a call to the main function, the instruction sequence in Salt also makes a call to the main function. The second premise holds: the evaluation context and the local environment in Lexi are both empty, and that corresponds to having a single empty stack in the data heap, and the stack pointer pointing to the bottom of the initial stack in Salt. The third premise holds: the data heap are both empty. The fourth premise holds: the code memory in Lexi and Salt are related by the definition of the program translation.

We now prove the second condition, that if $M_1 \sim M_2$, and M_1 is a final configuration, then M_2 is also a final configuration and their results are equal. Given that M_1 is a final configuration, the term in M_1 must be `halt` c. By the definition of the relation, the instruction sequence in M_2 must be a `halt` as well, and thus is also in a final configuration. To show that the results of the two configurations are equal, notice if M_1 is a `halt` c, it must have been reduced from `exit` v. The translation of `exit` v has already moved the value v to **r1**, which is where the result is stored in the final configuration of Salt.

To prove the third condition, we do a case analysis on the reduction rules of Lexi. Most of the rules are straightforward, and we focus on selected rules related to the handler constructs.

- **Case** HANDLE rule for general handler: The handler term in Lexi is translated to an instruction sequence that calls $\mathfrak{L}_{\text{handle}}$, which allocate a new stack and push the header frame of the handler. The control is then transferred to the body of the handler. We aim to prove that one step of the HANDLE rule is simulated by the execution of the aforementioned instruction sequence. We show that the resultant configurations M_1' and M_2' are related. M_1' has an extra activation frame on the old stack, and a handler frame on the new stack. This corresponds to the M_2' having an extra return address on the old stack, an extra stack in the heap, where the header frame at the bottom of the stack matches the handler frame in M_1'. In addition, the term in M_1' comes from the body of the handler, and this matches the instruction sequence in M_2', which is the body of the handler.
- **Case** HANDLE rule for tail-resumptive handler and abortive handler: omitted.
- **Case** Raise Handler rule: The raise handler rule in Lexi is translated to a sequence of instructions that calls $\mathfrak{L}_{\text{raise}}$, which allocates a one-element tuple and stores the pointer to the handler frame. The exchanger in the handler frame is set to the top of the current stack. The control is then transferred to the body of the handler. Now let's see how M_1' and M_2' are related. M_1' has a fresh resumption object in the data heap, and this matches the one-element tuple in M_2'. Part of the evaluation context in M_1 is moved to become the resumption in M_1'. This change is not directly reflected in M_2', because both the evaluation context before and the resumption after are related to the same stacks in M_2 and M_2'. The part that does change is the exchanger in the handler frame. But even with this change, the relation between M_1' and M_2' still holds, by the definition of the relation.

E AN INTERRUPTIBLE-ITERATOR PROGRAM

Figure 13 shows the Interruptible Iterator benchmark adapted from earlier work on bidirectional effects [13, 28].

```
effect Yield =
| yield : int List -> Replace -> Behead -> unit
effect Replace =
| replace : int -> unit
effect Behead =
| behead : unit -> unit

def client list =
    var beheaded := False
    val res_list =
        handle
            handle
                iterate list Y B
            with B
            | behead() => beheaded := True
        with Y
        | yield x R B =>
            if x < 0 then B.behead()
            else R.replace(x*2)
    if beheaded then res_list.drop(1)
    else res_list

def iterate list Y B =
    var hd := list.head()
    val tl = list.tail()
    var beheaded := False
    handle
        Y.yield(hd, R, B)
    with R
    | replace x => hd := x
    val newtl =
        if tl.isempty() then [] else
            handle
                iterate tl Y B
            with B
            | behead() => beheaded := True
    if beheaded then Cons(hd, drop(newtl, 1))
    else Cons(hd, newtl)
```

Figure 13. The Interruptible Iterator benchmark.