

Implementation Techniques for Lexical Effect Handlers

by

Cong Ma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Cong Ma 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

- External Examiner: Xavier Leroy
Professor, Institute of Mathematics and Numerical Sciences,
Collège de France
- Supervisor: Yizhou Zhang
Assistant Professor, David R. Cheriton School of Computer Science,
University of Waterloo
- Internal Member: Ondřej Lhoták
Professor, David R. Cheriton School of Computer Science,
University of Waterloo
- Internal Member: Khuzaima Daudjee
Research Professor, David R. Cheriton School of Computer Science,
University of Waterloo
- Internal-External Member: Werner Dietl
Associate Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis consists of material from three co-authored publications, of which I am the lead author and the major contributor.

- Chapter 2 is based on “Lexical Effect Handlers, Directly” [33], published at OOPSLA 2024 and co-authored with Zhaoyi Ge, Edward Lee, and Yizhou Zhang.
- Chapter 3 is based on “Virtualizing Continuations” [35], published at PLDI 2026 and co-authored with Jonghyun Jung and Yizhou Zhang.
- Chapter 4 is based on “Zero-Overhead Lexical Effect Handlers” [31], published at OOPSLA 2025 and co-authored with Zhaoyi Ge, Max Jung, and Yizhou Zhang.

Appendices A to C reproduce material from the technical reports [34, 36, 32] that accompany these publications. In accordance with the ACM publishing agreement, authors retain the right to reuse their published articles in a thesis or dissertation.

Abstract

Effect handlers generalize exception handling to support control-flow patterns including coroutines, cooperative multitasking, and nondeterminism. Recent work advocates for *lexically* scoped handlers, which restore abstraction safety and enable modular reasoning: a handler is a lexically scoped capability, and an effect can be raised only to a handler whose capability is in scope. However, prior implementations of lexical effect handlers suffer from inefficiencies that could hinder their adoption in practice.

This thesis shows that the static character that makes lexical scoping safe is also what makes it efficient to implement. It develops this insight into three complementary compilation techniques, embodied in our language LEXA; each technique is made precise by a formal model, proven correct, and realized in the LEXA compiler. Direct LEXA identifies each handler with the stack address where it is installed, so a raise jumps directly to its handler, without any search. Zone LEXA extends the address-based strategy to multishot resumptions by virtualizing handler identities through a software memory-management unit. Zero LEXA serves infrequent effects such as exceptions: it avoids the cost of passing down any run-time handler identity, and instead walks the stack using the static handler provenance information when an effect is raised, so effect-free execution pays nothing. A single program can choose among the three techniques per effect, matching each declared effect to its expected usage.

Together, these techniques demonstrate that lexical effect handlers can be implemented efficiently. Effect handlers can be both safe and fast.

Acknowledgements

My first impression of my advisor, Yizhou Zhang, was a sentence on his website: “I invite students to explore fun ideas in PL research.” It sounded a little too rosy to me at the time. But I have to say, the past four years were a truly fun ride. I am deeply indebted to Yizhou for his sharp technical insights and his openness to new ideas. This thesis is the culmination of the effort we poured into those ideas, and I hope some of the fun shows through in it.

My collaborators made significant contributions to the work in this thesis, and I am grateful for their help and support. Edward Lee helped me crack several tricky problems in the formalism of LEXA and was always a source of good conversation and encouragement. Zhaoyi (August) Ge wrote the initial implementation of LEXA and became a great friend through our outdoor adventures. Jonghyun (Max) Jung spearheaded the implementation of Zone LEXA and made early strides in formalizing its semantics. The projects in this thesis would have been far more difficult without collaborators as meticulous and hardworking as they are.

I thank the members of my thesis committee—Xavier Leroy, Ondřej Lhoták, Khuzaima Daudjee, and Werner Dietl—for their careful reading of this thesis and for their valuable feedback.

The members of the Programming Languages Group provided a supportive and stimulating environment. Ende Jin was always generous with his time in conversations about PL research and life in general. Jianlin Li was a great roommate and took care of my plants when I was away. Gaga Ebresafe’s enthusiasm for PL research was a pleasure to be around. Harris Lau, Jin Xu, Daisy Li, James You, Amir PashaeiHir, Pedro Oliveira, Alvin Zhang, Professor Ondřej Lhoták, Professor Peter Buhr, Professor Gregor Richards, and many others in the group were always helpful and supportive.

My other friends and colleagues at Waterloo made my life here all the more enjoyable. They include, in no particular order, Gengyi Sun, Ruizhe Wang, Emrick Donadei, Olha Tovstenko, Sparsh Sai Poosarla, Vanessa Phan, William Phan, and others.

Pittsburgh, my second home, also gave me friends I am grateful for. They include, in no particular order, Yifan Gu, Zhengyao Lin, Bing-Jui Ho, Xu He, Yi He, and others.

Finally, I thank my family. My wife, Jia Xia, has been my anchor through every high and low of this journey; her love made this thesis possible. And I thank my father and mother, whose unconditional support carried me here from the very beginning.

Dedication

To my family.

Table of Contents

Examining Committee Membership	ii
Author’s Declaration	iii
Statement of Contributions	iv
Abstract	v
Acknowledgements	vi
Dedication	vii
List of Figures	xii
List of Tables	xvi
1 Introduction	1
1.1 The Expressivity of Effect Handlers	2
1.2 Accidental Handling and Lexical Effect Handlers	7
1.3 Implementing Lexical Effect Handlers Efficiently	10
2 Lexical Effect Handlers, Directly	12
2.1 Lexical Effect Handlers: A Tale of Two Schedulers	13

2.2	Two Core Languages: LEXA and SALT	17
2.2.1	Source Language: LEXA	17
2.2.2	Target Language: SALT	22
2.3	Translating LEXA to SALT	25
2.3.1	Overview	25
2.3.2	Translating <code>handle</code> , <code>raise</code> , and <code>resume</code> with Trampolines	26
2.4	Optimizing Tail-Resumptive and Abortive Handlers	30
2.5	Correctness of the Translation	33
2.6	The LEXA Compiler	34
2.6.1	Overview	35
2.6.2	Limited Support for Multishot Resumptions	35
2.6.3	Stack-Switching Functions and Calling Convention	37
2.6.4	Reducing Allocations	40
2.7	Evaluation	40
2.8	Related Work	44
2.9	Summary	45
3	Zone LEXA: Virtualizing Continuations	47
3.1	The Multishot Challenge	47
3.2	Background: Handler Lookup in a Stack-Switching Runtime	52
3.3	Virtualizing Stack Space	55
3.4	A Formal Model of the Zone-Based Semantics	59
3.4.1	Syntax	59
3.4.2	Operational Semantics	59
3.5	Correctness	62
3.5.1	Relating SL and ZL	63
3.5.2	Type System	63
3.5.3	Proving Simulation	67

3.6	Implementation	69
3.6.1	Overview	69
3.6.2	Single-Shot and Last-Shot Optimizations	70
3.6.3	Optimizations for Abortive and Tail-Resumptive Handlers	72
3.6.4	Stack-Allocated Mutable State	73
3.6.5	Support for Simple Parallelism	73
3.6.6	Other Ways to Implement Virtual Resumptions	74
3.7	Evaluation	74
3.7.1	Overhead of Zone LEXA vs. Base LEXA	75
3.7.2	Speedup from Parallelizing Multishot Resumptions	77
3.7.3	Summary	78
3.8	Related Work	78
3.9	Conclusion	80
4	Zero LEXA: Zero-Overhead Lexical Effect Handlers	81
4.1	The Zero-Overhead Principle for Lexical Handlers	81
4.2	Main Ideas	83
4.2.1	Handler Search Semantics and Implementation Strategies	83
4.2.2	A Zero-Overhead Implementation Strategy	86
4.2.3	Non-Ambiguity Requirement on Capture Sets	89
4.3	A Formal Model of the Compilation Scheme	90
4.3.1	The Source Language SL	90
4.3.2	The Target Language TL	93
4.3.3	The Type-Directed Translation	97
4.4	Compiler Correctness	102
4.5	Implementation	107
4.5.1	Hopper Table	107
4.5.2	Stackwalker, Multishot Resumptions, and Tail-Resumptive Handlers	108

4.6	Evaluation	109
4.6.1	Effect-Heavy Benchmarks	109
4.6.2	Effect-Infrequent Benchmarks	111
4.6.3	Hopper Size and Stackwalking Time	111
4.7	Related Work	112
4.8	Conclusion	113
5	Conclusion	114
	References	116
	APPENDICES	122
A	Simulation Details for Direct LEXA	123
B	Full Formalism for Zone LEXA	131
B.1	Source Program Syntax	131
B.2	Runtime Syntax	131
B.3	Operational Semantics	132
B.4	Type System	133
B.5	Extension with Parallel-Let	135
C	Metatheory Details for Zero LEXA	136
C.1	Subject Reduction	136
C.2	Relating SL^+ and TL	139
C.2.1	Label and Capability Relation	139
C.2.2	Value Relation	141
C.2.3	Term Relation	141
C.2.4	Evaluation Context Relation	142
C.2.5	Continuation Relation	143
C.2.6	Configuration Relation	143
C.3	Simulation Proof	144

List of Figures

1.1	The same effect-raising operation, given two different interpretations by two different handlers.	3
1.2	Control inversion via an effect handler: a stock <code>iter</code> function is reused as a lazy generator. Pulling values from the generator implicitly resumes the paused walk over the list.	5
1.3	A cooperative scheduler with pause and fork.	6
1.4	Search algorithm using multishot resumptions.	7
1.5	Accidental handling under dynamic scoping.	8
1.6	Lexical scoping prevents accidental handling.	10
2.1	Lightweight cooperative multitasking via lexical effect handlers.	14
2.2	Scaling behaviour of the scheduler program implemented in Effekt and in LEXA.	15
2.3	Syntax of LEXA.	18
2.4	Syntax of the LEXA abstract machine.	19
2.5	Selected reduction rules of LEXA.	20
2.6	The remaining reduction rules of LEXA: local evaluation, heap access, function calls, and program exit.	21
2.7	Syntax of SALT.	23
2.8	Syntax of the SALT abstract machine.	23
2.9	Selected reduction rules of SALT.	24
2.10	Layout of a single stack in SALT.	25

2.11	Translation of selected LEXA constructs.	26
2.12	Built-in trampolines used by the translated code.	27
2.13	State transitions as control enters a handle expression and a handler instance is installed.	29
2.14	State transitions as an effect is raised, a resumption is captured, and control is transferred to a handler.	29
2.15	State transitions as a previously captured resumption is resumed.	29
2.16	Extending LEXA and the LEXA-to-SALT translation to address tail-resumptive and abortive handlers. The HANDLE and RAISE rules are omitted as they require minimal changes to the rules in Figure 2.5.	31
2.17	Stack layouts for tail-resumptive and abortive handlers, which place the header frame on the parent’s stack rather than on a freshly allocated stack.	32
2.18	The scheduler example in C using the StackTrek macros.	36
2.19	Stack-switching function for the RAISE macro.	38
2.20	For the program in (a), using setjmp and longjmp for stack switching can lead to stack overflow (b), while the preserve_none calling convention used by the LEXA compiler allows tail calls to be optimized (c).	39
2.21	Scaling behaviours of different systems on 14 benchmarks. Effekt is omitted from the Scheduler chart; it appears in Figure 2.2. Resume Nontail 2 and Interruptible Iterator are each represented by two charts; the second chart (with the title in bold) reports on the systems that exhibit super-linear scaling, using LEXA as a reference point.	43
3.1	Defining and solving stochastic search problems. The effect handlers use multishot resumptions.	49
3.2	A parallelized version (simplified) of the handler code for the choose and sample operations. This code leads to run-time errors in both Base LEXA and Effekt.	50
3.3	A program with five handlers. The implementations of the handlers, except e2 ’s, are elided. The variables e0–e4 are bound by the handle constructs and are available in the scope of the corresponding handled computations.	53
3.4	Stack snapshots at four program points during the execution of the program from Figure 3.3.	53

3.5	Stack snapshots before and after raising an effect within a copied resumption.	55
3.6	A stack snapshot with virtual resumptions. For clarity, stack segments are marked with handler IDs only; full virtual addresses can be reconstructed by prepending the zone number and appending the offset.	56
3.7	Structure of virtual addresses in the implementation described in Section 3.6. Address translation finds the actual zone number via zone walk.	56
3.8	Auxiliary syntax for the operational semantics.	60
3.9	Reduction rules of the SL and ZL semantics.	61
3.10	Type-level constructs and selected typing rules for values and expressions. .	65
3.11	Selected typing rules for handlers, frames, and stacks.	66
3.12	Layout of stack segments using OS virtual memory. Grey backgrounds represent zones. Black circles mark zone headers; the solid arrow links zone headers, while dashed arrows link consecutive stack segments.	70
3.13	Pseudocode for the StackTrek macros using the sMMU API.	71
3.14	Intra-zone header elision. Pink stack segments mark potential locations of an in-place resumption. If the resumption starts at a greater handler ID than that of the closest zone header (black circle), the intra-zone header can be elided.	72
3.15	Strong-scaling behaviour of Zone LEXA on embarrassingly parallel multishot benchmarks. Speedups are relative to each benchmark’s sequential baseline.	78
4.1	A program consisting of a library, a plugin, a framework, and an application. Code in pink pertains to capability polymorphism: framework ’s capability variable α , its instantiation in main , and the capture set $\{\alpha\}$ in the type of callback	84
4.2	Stack diagrams illustrating different handler-search semantics and implementation strategies.	85
4.3	Higher-order functions with assorted binding structures. Above each function body, the black boxes are the capability-variable binders, the grey boxes the label-variable binders, and the dashed white boxes the function’s capture set; dotted arrows indicate the lexical provenance of each variable.	87
4.4	Stack snapshots illustrating the stackwalker’s behaviour when effects are raised from different points of the program in Figure 4.3. Clues are shown as numbered circles.	88

4.5	Syntax of SL	92
4.6	Runtime syntax of SL	92
4.7	Operational semantics of SL as an abstract machine.	93
4.8	Syntax of TL , including its runtime forms.	95
4.9	Operational semantics of TL as an abstract machine.	96
4.10	Type-level constructs and auxiliary judgements of SL	99
4.11	The SL type system and type-directed translation: selected rules (part 1). . .	100
4.12	The SL type system and type-directed translation: selected rules (part 2). .	101
4.13	Definition of $\text{hopper}_H(C)$, where the call-site metadata is $H = T_0; \bar{T}; \bar{\ell} : \bar{F}$. .	101
4.14	Selected syntax of SL^+ . The difference between SL^+ and SL is highlighted. .	103
4.15	Compiling a hopper instance for the call site at line 10 of Figure 4.3: (a) the call-site information, (b) the resulting hopper as a table, and (c) its binary representation. For illustration, the tables assume that ℓ_0 and ℓ_1 carry distinct effect names F_0 and F_1 ; in Figure 4.3 itself both have the single effect name F	108
4.16	Running time of the Two Threads benchmark as a function of the time slice. As the slice grows, the Yield effect is raised less frequently, shifting the advantage from Direct LEXA to Zero LEXA.	112

List of Tables

2.1	Benchmark running times measured for different systems. “n/a” indicates that the benchmark could not be implemented in the respective system. For benchmarks that we could not compile with Effekt’s MLton back end, we use the fastest back ends that can handle them (Chez Scheme or Node.js). The fastest result for each benchmark is highlighted.	42
3.1	Benchmark running times (milliseconds, mean) for four implementations of lexical effect handlers. The first group of benchmarks does not involve multishot resumptions; the second group does. No parallelism among multishot resumptions is exploited by Zone LEXA in this table; RQ2 is addressed in Section 3.7.2. “n/a” marks benchmarks an implementation cannot run. Averages are taken over the benchmarks that Base LEXA supports.	76
4.1	Two benchmark sets: an existing suite of effect-heavy programs (top) and a new suite of effect-infrequent programs (bottom). Running times are in milliseconds. “Loss/Gain” is Zero LEXA relative to Direct LEXA; the last two columns give the hopper table as a percentage of binary size and stack-walking as a percentage of total running time. “–Inf” marks a benchmark that Direct LEXA optimizes to a constant time of 0. All experiments were conducted on a workstation with an AMD Ryzen 7 CPU (4.5 GHz).	110

Chapter 1

Introduction

Effect handlers [41, 42] are a language abstraction for structured, nonlocal control flow. They generalize exception handling: when a piece of code *raises* an effect, control transfers to a *handler* on the call stack. Handlers are functions that have one or more parameters, one of which is a *resumption* (also called a *continuation*) that represents the suspended computation. Resumptions are first-class values, and can be invoked zero or more times to resume the same suspended computation with a return value.

Effect handlers, as well as first-class continuations, are powerful abstractions that can express a wide variety of control-flow patterns. Although they have been studied in the programming languages research community for decades, they have only recently begun to see adoption in mainstream languages: OCaml has adopted them [48], Scala and WebAssembly are considering them [39, 40], and a growing number of research languages (among them Eff [2], Koka [24], Effekt [7], and LEXA [33]) are built around them. Lexical effect handlers in particular are beginning to draw interest beyond research languages: OxCaml [62] and Zig [56] are considering adopting them.

This thesis is about how to compile a particular flavour of effect handlers, *lexical* effect handlers, efficiently. The introduction is organized in three parts. Section 1.1 illustrates the expressive power of effect handlers through a series of examples. Section 1.2 then explains why conventional, *dynamically* scoped handlers compromise modular reasoning, a problem known as *accidental handling*, and how lexical effect handlers fix it. Section 1.3 then turns to the question this thesis answers: how to implement lexical effect handlers without paying a high price for their abstraction-safety benefits, and outlines the three techniques the following chapters develop.

1.1 The Expressivity of Effect Handlers

The expressivity of effect handlers comes from three sources. First, they *separate* the use of an effect from its interpretation: the same effect-raising code can be paired with different handlers to obtain different behaviours. Second, they support *nonlocal control transfer*: upon the raising of an effect, control is transferred to a handler that may be located at several hops up the call stack. Similarly, upon resuming a resumption, control is transferred back to the point where the effect was raised. Third, by allowing resumptions to be invoked multiple times, certain algorithms that require backtracking can be implemented elegantly.

We illustrate the expressivity with four examples that exploit the aforementioned features.

Example 1: Same operation, different policies. The same effect-raising code can be given completely different meanings by the handler installed by its caller. Consider a division operation that raises a `DivByZero` effect when the divisor is zero. Figure 1.1 shows two callers of the same `safeDiv`: one aborts with an error message, the other returns a default value of zero and continues.

The two handlers in Figure 1.1 produce different behaviours from the very same call `safeDiv 10 0`. `mainAbort`'s handler prints an error and does not use the resumption, so control exits the entire handle expression and goes to the end of `mainAbort`. `mainDefault`'s handler invokes `resume k 0`, so control returns to the raise site in `safeDiv` and the original raise expression `raise DivByZero ()` evaluates to 0. Neither behaviour is built into `safeDiv`; the policy lives entirely in the handler.

Example 2: Turning an iterator into a generator. An *iterator* is a function that walks a data structure and applies a caller-supplied function to each element; the iterator drives the traversal. A *generator* yields one element per call to the consumer; the consumer drives the traversal. The consumer is free to terminate the traversal or interleave it with another generator. The iterator and generator are dual ways of walking the same data: iterators are easier to implement, while generators are easier to use.

Iterators are easier to implement because the call stack records the current position in the walk, whereas the generator must reify this position as an explicit data structure. A list generator can get away with a single tail pointer, but a tree generator needs a stack of pending sub-walks.

```

1  effect DivByZero : unit -> int
2
3  def safeDiv (a: int) (b: int) : int =
4    if b == 0 then raise DivByZero ()
5    else a / b
6
7  def mainAbort () : unit =
8    handle
9      print (safeDiv 10 0)
10   with
11     | DivByZero () k =>
12       print "error: division by zero"
13
14  def mainDefault () : unit =
15    handle
16      print (safeDiv 10 0)
17   with
18     | DivByZero () k =>
19       resume k 0

```

Figure 1.1: The same effect-raising operation, given two different interpretations by two different handlers.

This is what makes *control inversion* both useful and interesting. By using effect handlers, an iterator can be turned into a generator, so the programmer can enjoy both the ease of implementation and the ease of use [47, 13, 58]. It is done by calling the iterator with a function that raises an effect for each element. When this happens, control is transferred to the handler, which receives the current element and the suspended traversal as a resumption. The consumer, which sits outside the handler, can then choose to resume the traversal to get the next element.

In a sense, the effect handler automatically reifies the position of the walk in a resumption, so the programmer does not have to build it by hand.

Figure 1.2 illustrates how the control inversion is achieved for a list iterator. The `iter` function is written exactly as one would expect: it walks the list and applies the function argument `f` to each element. `iterToGen` turns this into a generator by invoking `iter` with a function that raises `Yield` with each element. When this happens, the enclosing handler captures the suspended traversal as a resumption, stores it in the mutable cell `next`, and returns the value to the caller. The next call to the generator resumes that captured traversal until the next `Yield`.

The usefulness of this inversion is visible in `firstAbove30`: it stops pulling as soon as it finds an element exceeding 30, so the remaining elements of the list are never visited, even though the underlying `iter` is fundamentally eager and was not designed for early termination.

Example 3: Cooperative concurrency. The same ability to capture and resume continuations can be used to implement a cooperative scheduler as a library [33, 48]. As is shown in Figure 1.3, a task that wants to yield the CPU raises `Pause`, and a task that wants to spawn a child raises `Fork`. The scheduler functionality is implemented using two handlers for these effects. The `Pause` handler captures the current task and enqueues it. The `Fork` handler does that too, then runs the child task inside fresh handlers by calling `spawn`. An external `driver` repeatedly pops a paused task off the queue and resumes it.

Example 4: Backtracking search. The previous examples invoke the resumption at most once. Multishot resumptions, which can be invoked multiple times, are useful for implementing search algorithms [11], as shown in Figure 1.4. The variables `a`, `b`, and `c` are bound to return values of `Choose`, which is an effect that represents a nondeterministic choice among a list of integers. The handler resumes `k` once for every candidate value `x`

```

1  effect Yield : int -> unit
2
3  def iter (f: int -> unit) (xs: list[int]) : unit =
4    case xs of
5      | [] => ()
6      | x :: rest => f x; iter f rest
7
8  def iterToGen (xs: list[int]) : unit -> option int =
9    let next = ref (fun () =>
10      handle
11        iter (fun x => raise Yield x) xs;
12        none
13      with
14        | Yield x k =>
15          next := (fun () => resume k ());
16          some x
17    ) in
18    fun () => (!next) ()
19
20 def firstAbove30 (xs: list[int]) : option int =
21   let gen = iterToGen xs in
22   let rec loop () : option int =
23     case gen () of
24       | none => none
25       | some x => if x > 30 then some x else loop ()
26   in loop ()

```

Figure 1.2: Control inversion via an effect handler: a stock `iter` function is reused as a lazy generator. Pulling values from the generator implicitly resumes the paused walk over the list.

```

1  effect Pause : unit -> unit
2  effect Fork   : (unit -> unit) -> unit
3  effect Done   : unit -> 'a
4
5  def driver (q: queue) : unit =
6      handle
7          let k = if q.isEmpty then raise Done () else q.pop () in
8          resume k ();
9          driver q
10     with
11         | Done () _ => ()
12
13  def spawn (f: unit -> unit) (q: queue) : unit =
14      handle
15          f ()
16     with
17         | Pause () k => q.push k
18         | Fork g k =>
19             q.push k;
20             spawn g q
21
22  def scheduler (main: unit -> unit) : unit =
23      let q = newQueue () in
24      spawn main q;
25      driver q

```

Figure 1.3: A cooperative scheduler with pause and fork.

```

1 effect Choose : list[int] -> int
2
3 def triples (n: int) : list[(int, int, int)] =
4   handle
5     let a = raise Choose (range 1 n) in
6     let b = raise Choose (range a n) in
7     let c = raise Choose (range b n) in
8     if a*a + b*b == c*c then [(a, b, c)] else []
9   with
10    | Choose xs k =>
11      List.flatten (List.map xs (fun x => resume k x))

```

Figure 1.4: Search algorithm using multishot resumptions.

in the list, then concatenates the results. The entire body of `triples` effectively expands into a triply nested loop, computing all Pythagorean triples up to `n`.

Across the four examples, the same theme emerges: the control-flow mechanism offered by effect handlers achieves a separation of concerns. The iterator can be written where the control flow follows the shape of its data structure, making it easy to implement; the `Yield` handler then turns it into a generator abstraction, letting the consumer be written with a control flow that is convenient for its use. The scheduler is the same idea: each task is written as if it ran alone, and the `Pause`/`Fork` handler weaves the tasks together. Such separations of concerns are what make modular programs possible, and modularity is what lets software construction scale.

1.2 Accidental Handling and Lexical Effect Handlers

The modular promise of effect handlers is, in their conventional semantics, only partly delivered. Following the tradition established by exception handling [29], most languages with effect handlers make handlers *dynamically* scoped: when an effect is raised, the runtime walks the call stack and chooses the first handler it finds that matches the effect name. Dynamic scoping is intuitive and is the de facto choice for languages including OCaml [48] and Koka. But it also threatens modular reasoning [69, 68], for the same reason that dynamically scoped variables threaten modular reasoning and have fallen out of favour in general-purpose languages.

```

1  effect Logging : string -> unit
2
3  def framework1 (callback: unit -> unit) : unit =
4    handle
5      ...
6      callback ();
7      ...
8      raise Logging "framework: internal log"
9    with
10   | Logging msg k => writeToFile msg; resume k ()
11
12  def framework2 (callback: unit -> unit) : unit =
13    ...
14    callback ()
15    ...
16
17  def main () : unit =
18    handle
19      framework (fun () => raise Logging "from callback")
20    with
21   | Logging msg k => writeToConsole msg; resume k ()

```

Figure 1.5: Accidental handling under dynamic scoping.

Accidental handling. When handlers are dynamically scoped, it is possible for handlers to catch effects that are not meant for them. This is known as *accidental handling*. Such behaviour compromises abstraction safety, leaking implementation details and obstructing local reasoning about effectful programs.

Figure 1.5 illustrates the issue of accidental handling. Two implementations of a `framework` are shown, each of which takes in a callback and runs it. In `framework1`, the `Logging` effect is used to log internal messages, so the body of `framework1` is wrapped in a handler for `Logging`. In `framework2`, no `Logging` handler is installed.

`main` calls `framework` with a callback that raises a `Logging` effect. Now, consider the scenario where the programmer writing `main` initially uses `framework1`, and later replaces it with `framework2`. They will get different behaviours: in the first case, the callback's log message goes to a file, because the effect is handled by the `Logging` handler in `framework1` due to dynamic scoping; in the second case, the callback's log message goes to the console,

because the effect is handled by the `Logging` handler in `main`. Such a change in behaviour could be surprising to the programmer, because they cannot foresee it without knowing the implementation details of `framework`.

So what is the underlying problem here? A function like `framework` is a higher-order function: it takes another function as an argument. To make such a function reusable, it should be oblivious to the behaviour of the argument, which means that it should not have the ability to inspect or interfere with the effects raised by the argument. Readers might be familiar with a similar pattern in polymorphic functions like `map`, which operates on lists of any type. If `map` is implemented in a way that inspects the type of the elements, then it will not be reusable for all types, as the behaviour of `map` might be different when the elements are integers or strings.

The dynamic scoping of handlers, and the resulting accidental handling, result in a violation of abstraction safety, analogous to a loss of parametricity, first recognized by Zhang and Myers [68]. It compromises the ability to reason about code locally, and becomes a barrier to the adoption of effect handlers in practice.

Lexical effect handlers. *Lexical effect handlers* solve this problem by treating each handler as a lexically scoped capability [69, 68, 7, 4, 65, 33]. Installing a handler binds a fresh capability; only code that holds this capability (either because it is textually inside the handler’s scope or because the capability has been passed to it explicitly) can raise effects to that handler.¹ There is no search up the call stack; the choice of handler is statically determined by which capability is used at the raise site.

Figure 1.6 rewrites the previous example using lexical effect handlers. The callback now explicitly raises to `consoleLogger`, a capability bound in `main`. The framework’s internal `fileLogger` is reified to a *different* capability; it cannot intercept an effect addressed to `consoleLogger`. Whichever framework `main` happens to use, the callback’s log goes to the console. Local reasoning is restored: to know where a raise will be handled, one needs only to follow the binding of the capability that appears at the raise site.

Adoption. Languages that adopt lexical effect handlers in some form include Genus [69], Effekt [7], and our own LEXA [33]. Koka’s *named handlers* [65] and proposals for WebAssembly [40]

¹Later chapters also use the term *label* for the fresh name, generated at run time, that identifies a handler instance. Labels and capabilities are closely related notions but play different roles: a label is the identity of a handler, while a capability is the right to raise effects to it. This thesis will use the terms *label*, *capability*, *handler reference*, and *handler identity* interchangeably in the prose, but the formalism will make the notions precise.

```

1 effect Logging : string -> unit
2
3 def framework1 (callback: unit -> unit) : unit =
4   handle (fileLogger: Logging) =>
5     callback ();
6     raise fileLogger "framework: internal log"
7   with  $\lambda$  msg k =>
8     writeToFile msg; resume k ()
9
10 def main () : unit =
11   handle (consoleLogger: Logging) =>
12     framework1 (fun () => raise consoleLogger "from callback")
13   with  $\lambda$  msg k =>
14     writeToConsole msg; resume k ()

```

Figure 1.6: Lexical scoping prevents accidental handling.

provide closely related mechanisms. The formal payoff has been mapped out in several papers [68, 4]: lexical scoping recovers abstraction-safety guarantees that dynamic scoping compromises, while preserving the expressive power of effect handlers.

1.3 Implementing Lexical Effect Handlers Efficiently

Adopting lexical effect handlers is attractive in principle, but only if they can be implemented efficiently in practice. The implementation challenge can be seen from its semantics: lexical scoping demands that, at every `raise`, the program locates the correct handler instance according to the lexical provenance of the capability. This is very different from dynamic scoping, where the program simply walks the call stack until it finds a handler with the matching name. Existing implementations prior to the work of this thesis incur overhead that hinders adoption in practice. Genus [69] implements lexical handlers by piggybacking them on dynamically scoped handlers and inheriting their search overhead; Effekt [7] uses CPS transformation and in some cases incurs a search behaviour similar to that of dynamically scoped handlers.

The remainder of this thesis presents three implementation techniques that, together, demonstrate that the abstraction-safety benefits of lexical effect handlers need not come at a performance cost. The three build on one another, and the later two each confront a

challenge left open by the last. Each chapter is based on a published paper of the author.

- Chapter 2 (Direct LEXA) confronts the most immediate challenge: the run-time *search* for a handler. Lexical scoping fixes which handler instance an effect must reach, but a naive implementation still scans the call stack to find it at run time, just as dynamic scoping does. This search is the overhead that has held back existing implementations. Direct LEXA, which implements lexical effect handlers in a stack-switching runtime, identifies every handler with the stack address where it is installed. That address is passed down to the raise site as an ordinary argument, so raising an effect transfers control *directly* to its handler, and the resumption is captured *directly* as a span of stack, with no walking. This chapter is based on “Lexical Effect Handlers, Directly” [33], published at OOPSLA 2024.
- Chapter 3 (Zone LEXA) confronts the challenge that the address-as-identity trick breaks under *multishot* resumptions. Resuming a continuation more than once requires duplicating the captured stack segments, yet the copies inherit handler references that still point into the original stack, so a stack address no longer uniquely identifies a handler. Direct LEXA simply forbids the use of multishot resumptions. Zone LEXA relaxes this restriction by introducing a novel technique called *virtual resumption*, which makes handler references *virtual* and translates them to the correct copy of the handler on demand, through a software memory-management unit modelled on the virtual memory of operating systems. This chapter is based on “Virtualizing Continuations” [35], published at PLDI 2026.
- Chapter 4 (Zero LEXA) confronts a challenge the previous two leave open: giving each handler a run-time identity is not free, because the identity must be reified and threaded down the call chain. For the large class of programs that use exception handlers, the cost incurred by the parameter passing is non-negligible, and it is paid for even when these handlers are never invoked. Zero LEXA restores the *zero-overhead principle* (what you do not use, you do not pay for) by avoiding handler reification altogether: it instead spreads handler provenance information across the call stack as static metadata, and relies on stackwalking to locate the intended handler at run time. This eliminates the run-time overhead on the mainline code when handlers are not used. This chapter is based on “Zero-Overhead Lexical Effect Handlers” [31], published at OOPSLA 2025.

Chapter 2

Lexical Effect Handlers, Directly

Chapter 1 argued that lexical effect handlers restore the modular-reasoning principles that dynamically scoped handlers compromise, and it ended by posing the implementation question: can lexical handlers be compiled efficiently? This chapter answers the question in the affirmative. We present a compilation strategy that targets a low-level language with control over stack layout. We embody the ideas in a language called LEXA and its compiler, and run an evaluation on a suite of benchmarks; the results suggest that it generates efficient code, in some cases reducing running-time complexity from quadratic to linear. This chapter is based on “Lexical Effect Handlers, Directly” [33], published at OOPSLA 2024.

In LEXA, the low-level representation of a handler is no different from that of any lexically scoped local variable: it is a stack location where the handler is installed. This location is passed down the call stack as an argument, eventually reaching the point where the effect is raised. When an effect is raised, the control-transfer target is *directly* available—no run-time search is needed to identify the handler. In addition, the suspended computation (a.k.a. the resumption) is captured *directly* as stacks, again without any walking. The idea is natural in the sense that it is hinted at by the connection between lexical effect handlers and multi-prompt delimited control [20]. Prior work suggests a connection in semantics [68, 8, 70]; this work shows that the connection extends to implementation. Our approach is reminiscent of native implementations of multi-prompt delimited control such as libmprompt [25], while incorporating optimizations not present in those implementations.

The chapter proceeds as follows. Section 2.1 reviews the design of lexical effect handlers through a scheduler example and identifies a source of inefficiency in existing implemen-

tations. Section 2.2 defines two core languages: LEXA, an intermediate language with lexical effect handlers, and SALT, an assembly-like language with control over stack layout. Section 2.3 describes the translation from LEXA to SALT, and Section 2.4 describes optimizations for tail-resumptive and abortive handlers. Section 2.5 establishes the correctness of the translation. Section 2.6 describes the implementation of the LEXA compiler, and Section 2.7 presents an empirical evaluation. Section 2.8 discusses related work, and Section 2.9 concludes.

2.1 Lexical Effect Handlers: A Tale of Two Schedulers

We use the cooperative multitasking scheduler in Figure 2.1 to review the ideas of lexical effect handlers and to expose the inefficiency that this chapter sets out to eliminate. The example, adapted from prior work [48], is written in an OCaml-like syntax whose type system, like OCaml’s, does not track effects. It would be straightforward to layer a type-and-effect system on top; the problem is well studied [69, 68, 4, 7, 70] and orthogonal to the focus of this chapter.

Process is an *effect signature* containing two *effect operations*, **yield** and **fork**. The scheduler uses an *effect handler* to interpret these operations. It operates on a queue of resumptions, initially empty. The function **spawn** starts a computation **f** of type **Process** $\rightarrow$ **unit**; as **f** may raise **Process** effects, **spawn** installs a handler for them. Raising **yield** suspends the current job and hands control back to the scheduler; raising **fork** additionally asks the scheduler to run a new job. Specifically, when **yield** is raised, the remaining computation in the **handle** body is captured as a resumption **k**, which is enqueued 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 resumptions. **driver** dequeues a resumption, resumes it (line 12), and recursively calls itself to run the next one. When the queue is empty, the call to **dequeue** raises an exception (via the **Exn** signature), and **driver** handles it by logging a message before exiting.

Handlers in this example are *lexical*. A **handle** expression binds a variable representing the handler in its body. For instance, the **handle** expression in **spawn** binds a handler named **P** (line 19), which is then passed as an argument to **f** to handle the **Process** effects raised by **f** (line 20). A **raise** expression explicitly mentions the handler to which the effect is raised; for instance, line 40 raises the **fork** effect to the handler named **P**. Although all uses of handlers are explicitly named here, prior work has shown that explicit names are not necessary in many cases: a lighter-weight syntax is possible by resolving

```

1  effect Process =
2  | yield : unit -> unit
3  | fork  : (Process -> unit)
4           -> unit
5
6  effect Exn =
7  | throw : 'a
8
9  def driver q =
10   handle (E: Exn) =>
11     val k = dequeue E q;
12     resume k ();
13     driver q
14   with
15   | throw k =>
16     log "all resumptions done"
17
18  def spawn (f: Process -> unit) q =
19   handle (P: Process) =>
20     f P
21   with
22   | yield _ k => enqueue k q
23   | fork g k =>
24     enqueue k q; spawn g q
25
26  def scheduler (f: Process -> unit) =
27   val q = mkQueue ();
28   spawn f q;
29   driver q
30
31  def job P =
32   ... raise P.yield (); ...
33
34  effect Tick =
35  | tick : unit
36
37  def jobs P T n =
38   if n == 0 then ()
39   else
40     raise T.tick;
41     raise P.fork job;
42     jobs P T (n - 1)
43
44  def main () =
45   val c = ref 0;
46   handle (T: Tick) =>
47     scheduler (fun P =>
48       jobs P T 1000)
49   with
50   | tick k => c := !c + 1;
51   resume k ()

```

Figure 2.1: Lightweight cooperative multitasking via lexical effect handlers.

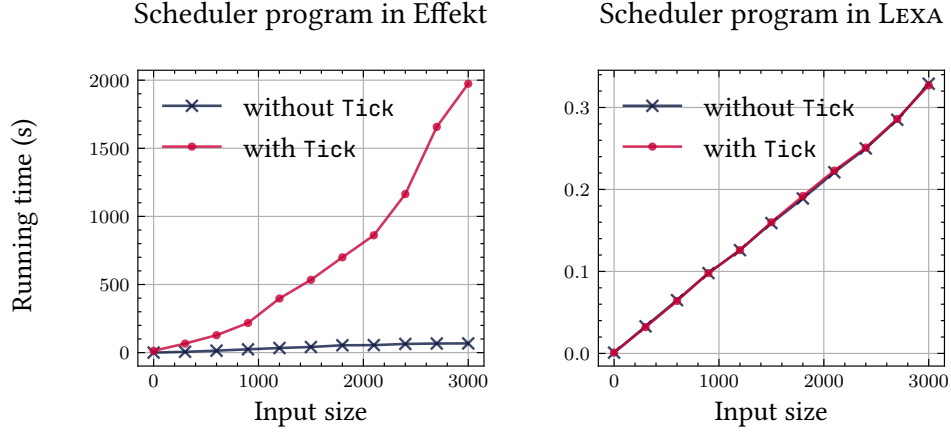


Figure 2.2: Scaling behaviour of the scheduler program implemented in Effekt and in LEXA.

omitted handler annotations to handler bindings in the lexical context via a straightforward translation [68, 7].

The semantics of lexical effect handlers exhibits similar behaviour to that of the *deep handler* semantics [23]: when an effect is raised to a handler, the captured resumption contains the very handler in its outermost layer. Thus, at line 12, the resumed computation may raise `yield` or `fork` again, and the handler that was installed earlier in the resumption will handle those effects.

The `main` function uses `scheduler` to run a `jobs` function, which spawns 1000 jobs by raising `fork` effects. Each job may voluntarily `yield` to the scheduler. In addition, the programmer of `main` wants to keep count of how many times `fork` has been raised, which is done by raising a `Tick` effect right before each `fork` (line 39). The `main` function handles `Tick` by incrementing a counter.

The costly tick. It is believed that Effekt [7, 45, 38], a language that prominently features lexical effect handlers, does not require a run-time search for handlers when an effect is raised. So raising and handling effects should be cheap, comparable to function calls, and the running time of the program should be proportional to the number of jobs. We created a similar scheduler program in Effekt. Figure 2.2 shows how the running time scales. Surprisingly, we observe super-linearly increasing running time. Interestingly, if the `Tick` effect is turned off, linear scaling is restored. The seemingly innocuous `Tick` effect

drastically changes the performance of the program.¹

Why are the **Tick** effects so expensive for this scheduler implemented in Effekt? This experiment uses a back end of the Effekt compiler that translates control effects into a monadic framework for multi-prompt delimited control [14]. When an effect is raised, a sequence of prompt markers—each corresponding to a handler in the surrounding evaluation context—is traversed to find the right handler. In the scheduler program, a **Tick** effect is propagated through the entire call chain to the handler in **main**. The call chain, which is deep due to the recursive nature of **driver**, happens to contain one **Exn** handler per level of recursion. So the asymptotic time complexity of handling a single **Tick** effect is $O(n)$, where n is the number of jobs forked; the total running time of the program is therefore $O(n^2)$.

Effekt also has a back end that first performs a *lift inference* [38] before a CPS translation to MLton. Unfortunately, this back end does not support the scheduler program, due to a conflict between MLton’s monomorphization requirement and the typed CPS translation. But even if the program could be compiled, it would not scale well: the lift inference generates a series of function applications that, at run time, effectively traverse the surrounding evaluation context to find the right handler for an effect.

Hence, both of these compilation strategies share a common inefficiency. Although there is no run-time search of the *call stack* for handlers—there is no call stack to begin with, as these back ends do not need or manipulate call stacks—raising and handling an effect incurs a cost similar to that of a stack walk. We consider these compilation strategies *lift-based*: they employ run-time computations to lift an effect out of the dynamic extents of handlers until the right handler is found.

Quadratic scaling can be avoided in this particular program by carefully adjusting the lexical scope of the exception handler so that only one is installed in the evaluation context at any time, but it would be ideal if the programmer did not have to concern themselves with such details. Furthermore, there exist useful programs for which deep stacks of handlers are unavoidable—we will see one in Section 2.7—and for those programs the scalability issue cannot be addressed merely by hand-tuning the scopes of handlers.

To lift, or not to lift? A lift-based approach is perhaps necessary for a compiler targeting a high-level language, but one may wonder whether the run-time cost is avoidable for a compiler targeting a language that allows low-level control over the call stack. In a certain

¹The most recent version of Effekt has since adopted the approach laid out in this chapter and no longer exhibits this overhead [37].

sense, lift goes against the grain of lexical handlers. The lift construct of Biernacki et al. [3] is a related language mechanism introduced to tame dynamically scoped handlers so that they do not compromise abstraction safety. The abstraction-safety problem does not arise in the context of lexical handlers [68], so lift ought to be unnecessary. This chapter offers a strategy to compile lexical handlers without lift. As Figure 2.2 shows, the running time of the scheduler implemented in LEXA scales linearly, with or without Tick effects.

2.2 Two Core Languages: LEXA and SALT

We formalize the key aspects of compiling lexical effect handlers to a stack-based low-level language using two core languages, LEXA and SALT. LEXA is a compiler intermediate language with support for lexical handlers, and SALT is an assembly-like language with support for low-level control over memory and stack layout. Sections 2.3 to 2.5 will define a translation from LEXA to SALT and prove it correct. We use the two languages to capture the core ideas of how we compile lexical handlers; it is not a goal to formally verify a realistic compiler. Our compiler implementation, described in Section 2.6, largely follows the formalism and supports additional features and optimizations.

2.2.1 Source Language: LEXA

LEXA programs use lexically scoped variables to identify handlers. LEXA is intended as a compiler intermediate language, where every use of a handler explicitly refers to the variable that binds the handler in the lexical context. A surface language can provide a lighter-weight syntax by resolving uses of handlers to variables in scope, as formalized in prior work [68, 7]. To simplify the formalization, LEXA programs are assumed to have undergone closure conversion and hoisting, so all functions, including those representing handlers, are closed and hoisted to the top level. Our compiler implementation does include these standard transformations.

Because effect handlers in LEXA have a unified representation as functions, LEXA supports *bidirectional effects* [70], namely the ability for a handler to raise reverse-direction effects to the computation that raised the initial effect. Similarly to Multicore OCaml [48], which also compiles to call stacks, it is not a goal of the formalism to support handlers with multishot resumptions; thus, in the LEXA formalism, a resumption can be resumed at most once. Our compiler implementation does have limited support for multishot resumptions (Section 2.6.2). Full support for multishot resumptions will be described in

constant	c	$::=$	$i \mid P \mid \mathbf{ns}$
value	v	$::=$	$x \mid c$
expression	e	$::=$	$v \mid v_1 + v_2 \mid \mathbf{newref} (v_0, \dots, v_{n-1}) \mid \pi_i(v) \mid v_1[i] \leftarrow v_2 \mid v(v_1, \dots, v_n) \mid$ $\mathbf{handle} P_{\text{body}} \mathbf{with} P_{\text{op}} \mathbf{under} v_{\text{env}} \mid \mathbf{raise} v_1 v_2 \mid \mathbf{resume} v_1 v_2 \mid \mathbf{exit} v$
term	t	$::=$	$v \mathbf{end} \mid \mathbf{let} x = e \mathbf{in} t$
program	G	$::=$	$\mathbf{letrec} P_1 = \lambda \bar{x}_1. t_1, \dots, P_n = \lambda \bar{x}_n. t_n$

Figure 2.3: Syntax of LEXA.

Chapter 3. As an intermediate language, LEXA is untyped, but it would be possible to extend it with a type-and-effect system to ensure type, effect, and abstraction safety as in prior work [68, 70].

Syntax. Figure 2.3 presents the syntax of LEXA. *Expressions* are in A-normal form [44, 17], where every sub-expression is a variable or a constant. *Terms* are sequences of expressions: the term $\mathbf{let} x_1 = e_1 \mathbf{in} \dots \mathbf{let} x_n = e_n \mathbf{in} v \mathbf{end}$ sequences the expressions $e_1, \dots, e_n$, binding x_i to e_i in the rest of the term. We sometimes elide the trailing $\mathbf{end}$ for brevity. The notation $|t|$ denotes the number of $\mathbf{let}$ bindings in a term t . The metavariable P ranges over *code labels*, which identify functions (including implementations of effect operations) in the program text. A *value* is a local variable x , an integer i , a code label P , or nonsense $\mathbf{ns}$. An *expression* takes one of the following forms: a value, an arithmetic operation, allocating a tuple, reading or writing a tuple, applying a function, installing a handler, raising an effect to an installed handler, resuming a resumption, and exiting the program. In the expression $\mathbf{handle} P_{\text{body}} \mathbf{with} P_{\text{op}} \mathbf{under} v_{\text{env}}$, P_{op} is the label of the handler code (the implementation of the effect operation), P_{body} is the label of the handled code, and v_{env} provides the closure environment for both. A *program* consists of a sequence of top-level, possibly mutually recursive functions, one of which is designated as the main function. Effect handlers are defined as functions, with the last argument being the resumption. As a standard simplification, the formalism assumes that each handler has exactly one operation and that each operation has exactly one argument; the compiler is free of these restrictions.

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

Figure 2.4: Syntax of the LEXA abstract machine.

Operational semantics. We give an operational semantics to LEXA as an abstract machine. Figure 2.4 shows the extra syntax needed for the machine. Terms are extended with a run-time form $\mathbf{halt}$, which the EXIT rule steps to and which the final-configuration predicate of Section 2.5 recognizes. The 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*, generated by evaluating $\mathbf{newref}$, and *handler labels*, generated by evaluating $\mathbf{handle}$. A machine state (a *configuration*) consists of an immutable code memory, a mutable heap, an evaluation context, a local environment, and a redex. A code memory M maps code labels to functions. A heap H maps object labels to *heap values*; a heap value is either a tuple $\langle v_0, \dots, v_{n-1} \rangle$ or a resumption $\mathbf{cont } K$. An evaluation context K is a sequence of *frames* F . A frame can be an *activation frame* $(E, \mathbf{let } x = \square \mathbf{ in } t)$, which pairs a local environment E with a continuation expecting a value from the hole $\square$; or a *handler frame* $\#L^{P_{\text{op}}, L_{\text{env}}} \square$, which records a label L identifying the handler instance, a label P_{op} identifying the handler code, a label L_{env} identifying the closure environment, and a hole $\square$. A local environment E records the values of arguments and local variables for an activation of a function.

Figure 2.5 shows the rules that allocate on the heap or involve effect handling; Figure 2.6 shows the remaining rules. Since the syntax is in A-normal form, the order of evaluation is already determined, so there are no structural rules for locating the next redex. The rules fall into two groups. Rules in the first group evaluate an expression locally; this group includes $\mathbf{NEWREF}$, which allocates a 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 the second group involve control-flow

NEWREF

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

where L is fresh

HANDLE

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

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

LEAVE

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

RAISE

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

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

RESUME

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

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

Figure 2.5: Selected reduction rules of LEXA.

VALUE

$$\langle M \parallel H \parallel K \parallel E \parallel \text{let } x = v \text{ in } t \rangle \longrightarrow \langle M \parallel H \parallel K \parallel E[x \mapsto \hat{E}(v)] \parallel t \rangle$$

ADD

$$\langle M \parallel H \parallel K \parallel E \parallel \text{let } x = v_1 + v_2 \text{ in } t \rangle \longrightarrow \langle M \parallel H \parallel K \parallel E[x \mapsto \hat{E}(v_1) + \hat{E}(v_2)] \parallel t \rangle$$

GET

$$\langle M \parallel H \parallel K \parallel E \parallel \text{let } x = \pi_i(v) \text{ in } t \rangle \longrightarrow \langle M \parallel H \parallel K \parallel E[x \mapsto c_i] \parallel t \rangle$$

where $0 \leq i < n$, $\hat{E}(v) = L$, and $H(L) = \langle c_0, \dots, c_{n-1} \rangle$

SET

$$\langle M \parallel H \parallel K \parallel E \parallel \text{let } x = v[i] \leftarrow v' \text{ in } t \rangle \longrightarrow \langle M \parallel H[L \mapsto \langle c_0, \dots, \hat{E}(v'), \dots, c_{n-1} \rangle] \parallel K \parallel E[x \mapsto \hat{E}(v')] \parallel t \rangle$$

where $0 \leq i < n$, $\hat{E}(v) = L$, and $H(L) = \langle c_0, \dots, c_{n-1} \rangle$; the i th element is replaced

APP

$$\begin{aligned} &\langle M \parallel H \parallel K \parallel E \parallel \text{let } x = v(v_1, \dots, v_n) \text{ in } t \rangle \longrightarrow \\ &\langle M \parallel H \parallel K \cdot (E, \text{let } x = \square \text{ in } t) \parallel [x_1 \mapsto \hat{E}(v_1), \dots, x_n \mapsto \hat{E}(v_n)] \parallel t' \rangle \end{aligned}$$

where $\hat{E}(v) = P$ and $M(P) = \lambda(x_1, \dots, x_n).t'$

RET

$$\langle M \parallel H \parallel K \cdot (E, \text{let } x = \square \text{ in } t) \parallel E' \parallel v \text{ end} \rangle \longrightarrow \langle M \parallel H \parallel K \parallel E[x \mapsto \hat{E}'(v)] \parallel t \rangle$$

EXIT

$$\langle M \parallel H \parallel K \parallel E \parallel \text{let } y = \text{exit } v \text{ in } t \rangle \longrightarrow \langle M \parallel H \parallel K \cdot (E, \text{let } y = \square \text{ in } t) \parallel [x_{\text{result}} \mapsto \hat{E}(v)] \parallel \text{halt} \rangle$$

Figure 2.6: The remaining reduction rules of LEXA: local evaluation, heap access, function calls, and program exit.

transfer: APP, RET, HANDLE, LEAVE, RAISE, RESUME, and EXIT. They push or pop frames and switch to a different local environment. We focus on the four rules governing effect handling.

The HANDLE rule pushes an activation frame (the current local environment and the remaining term) and a handler frame onto the evaluation context. The handler frame contains a freshly generated handler label L that identifies the newly installed handler instance. The rule then sets up a new local environment for P_{body} , binding x_{env} to the closure environment L_{env} and x_{hdl} to the handler label L , and runs the body. The generativity of L matches the semantics of previous languages supporting lexical handlers, such as Genus [69] and Effekt [7].

The LEAVE rule pops the handler frame and returns to the most recent activation frame, updating the local environment with the return value.

The RAISE rule suspends the current computation and transfers control to a handler. The first operand of **raise** is interpreted into a handler label L identifying the handler instance to raise to; the second is the argument to the effect operation. A handler frame matching L is found in the evaluation context, and the context is unwound to that point. A resumption, made up of the unwound frames, is reified and stored in the heap; a fresh label L_k identifies it and can be used as a first-class value. The handler code P_{op} becomes the next redex, receiving its closure environment, the operation argument, and the resumption L_k .

The RESUME rule continues a suspended computation. The first operand of **resume** is interpreted into a label L_k identifying the resumption; the frames stored under L_k are pushed back onto the evaluation context, the suspended local environment is restored and updated with the resumption argument, and the suspended term resumes. To prevent the resumption from being resumed again, the heap is updated to map L_k to the nonsense value **ns**; attempting to use L_k again causes the machine to get stuck. This enforces, dynamically, that a resumption is resumed at most once.

2.2.2 Target Language: SALT

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

Syntax. Figure 2.7 presents the syntax of SALT. A *word* is an address ℓ , an integer i , or nonsense **ns**. An *operand* is a word v or a register name **r**. *Instructions* are standard for an abstract assembly language; the only exception is the **mkstk** instruction, which allocates a

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

Figure 2.7: Syntax of SALT.

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

Figure 2.8: Syntax of the SALT abstract machine.

new stack on the heap. A *program* G is a finite map from code addresses (ranged over by P) to instruction sequences, one of which is designated as the main function.

Operational semantics. We give an operational semantics to SALT as an abstract machine; the extra syntax needed is shown in Figure 2.8. A *stack* is a sequence of words ending with a special symbol $\mathbf{nil}$ that marks the *base* (as opposed to the *top*) of the stack. Stacks are heap-allocated; they are heap values, just like tuples. An *address* can be a heap location L or a code-memory location P ; heap locations cannot appear in the program text. Locations can be constructed through the $\mathbf{next}$ constructor; the notations ℓ^i and $\mathbf{next}^i(\ell)$ are interchangeable. For example, if L is the base of a stack in a heap H , meaning $H(L) = \mathbf{nil} :: v_1 :: \dots :: v_n$, then $L^j = \mathbf{next}^j(L)$ is the stack location where v_j is stored.

$$\begin{array}{c}
\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \mathbf{mkstk} \ \mathbf{r} \quad L \text{ is fresh}}{\langle M \parallel H \parallel R \rangle \longrightarrow \langle M \parallel H[L \mapsto \mathbf{nil}] \parallel R[\mathbf{ip} \mapsto \mathbf{next}(\ell), \mathbf{r} \mapsto L] \rangle} \\
\\
\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \mathbf{salloc} \ n \quad R(\mathbf{sp}) = L^m \quad H(L) = \mathbf{nil} :: v_1 :: \dots :: v_m}{\langle M \parallel H \parallel R \rangle \longrightarrow \langle M \parallel H[L \mapsto \mathbf{nil} :: v_1 :: \dots :: v_m :: \underbrace{\mathbf{ns} :: \dots :: \mathbf{ns}}_n] \parallel R[\mathbf{ip} \mapsto \mathbf{next}(\ell), \mathbf{sp} \mapsto L^{m+n}] \rangle} \\
\\
\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \mathbf{mov} \ \mathbf{sp}, o \quad \hat{R}(o) = L^m \quad H(L) = \mathbf{nil} :: v_1 :: \dots :: v_m :: \dots :: v_n}{\langle M \parallel H \parallel R \rangle \longrightarrow \langle M \parallel H[L \mapsto \mathbf{nil} :: v_1 :: \dots :: v_m] \parallel R[\mathbf{ip} \mapsto \mathbf{next}(\ell), \mathbf{sp} \mapsto L^m] \rangle} \\
\\
\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \mathbf{call} \ o \quad R(\mathbf{sp}) = L^m \quad H(L) = \mathbf{nil} :: v_1 :: \dots :: v_m \quad \hat{R}(o) = \ell_{\text{dest}}}{\langle M \parallel H \parallel R \rangle \longrightarrow \langle M \parallel H[L \mapsto \mathbf{nil} :: v_1 :: \dots :: v_m :: \mathbf{next}(\ell)] \parallel R[\mathbf{ip} \mapsto \ell_{\text{dest}}, \mathbf{sp} \mapsto L^{m+1}] \rangle} \\
\\
\frac{R(\mathbf{ip}) = \ell \quad \hat{M}(\ell) = \mathbf{return} \quad R(\mathbf{sp}) = L^m \quad H(L) = \mathbf{nil} :: v_1 :: \dots :: v_{m-1} :: \ell_{\text{dest}}}{\langle M \parallel H \parallel R \rangle \longrightarrow \langle M \parallel H[L \mapsto \mathbf{nil} :: v_1 :: \dots :: v_{m-1}] \parallel R[\mathbf{ip} \mapsto \ell_{\text{dest}}, \mathbf{sp} \mapsto L^{m-1}] \rangle}
\end{array}$$

Figure 2.9: Selected reduction rules of SALT.

A *register file* R contains finitely many registers, including the distinguished stack-pointer register $\mathbf{sp}$ and instruction-pointer register $\mathbf{ip}$; we write $\hat{R}(\mathbf{r}) = R(\mathbf{r})$ and $\hat{R}(v) = v$. A configuration C consists of an immutable code memory, a mutable heap, and a mutable register file.

Figure 2.9 shows selected rules that deal with stacks. To take a step, the instruction at the address in $\mathbf{ip}$ is executed. The register $\mathbf{sp}$ always points to the top of a stack—an invariant respected by all rules. A $\mathbf{mkstk}$ instruction allocates a new stack; $\mathbf{salloc}$ (resp. $\mathbf{sfree}$) grows (resp. shrinks) the stack pointed to by $\mathbf{sp}$, with new slots initialized to $\mathbf{ns}$. For $\mathbf{mov} \ \mathbf{r}_d, o$, the word $\hat{R}(o)$ is copied into $\mathbf{r}_d$; in case $\mathbf{r}_d$ is $\mathbf{sp}$ and $\hat{R}(o)$ is a stack location L^m , the stack L is cut so that $\mathbf{sp}$ points to the top L^m . The opcodes $\mathbf{call}$ and $\mathbf{return}$ are like those in x86: they push or pop the return address onto or off the stack.

Notice that SALT has no instructions specialized to effect handlers; handlers are compiled to its low-level, general-purpose instructions. This low-level nature distinguishes our approach from previous work [64, 66, 45] that defines translations to formal models of high-level functional languages à la System F.

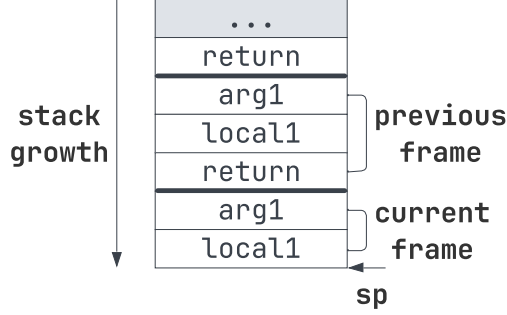


Figure 2.10: Layout of a single stack in SALT.

2.3 Translating LEXA to SALT

2.3.1 Overview

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

Figure 2.11 defines the translation for selected LEXA constructs. The translation of a function uses a simplified calling convention where all arguments are passed through registers and, upon entering a function, immediately pushed onto the stack. Register **r1** is used to store the return value. Before leaving a function, the callee deallocates its stack frame, and the final **return** instruction pops the return address and jumps to it.

The layout of a single stack is shown in Figure 2.10. The stack grows downwards. A black bar marks the base of a stack, and a thick black line marks the boundary between frames. The stack consists of a sequence of frames, each starting with the arguments and local variables and ending with the return address; the most recent frame has no return address. The translation of a term leaves the result of evaluating the term in **r1**; the translation of a value v stores its word into a provided register. Most cases of the expression translation are ordinary, except for **handle**, **raise**, and **resume**.

$\llbracket G \rrbracket = G$	$\llbracket e \rrbracket_\Gamma = I$
$\llbracket \text{letrec } P_1 = \lambda \overline{x_1}. t_1, \dots, P_n = \lambda \overline{x_n}. t_n \rrbracket =$ $P_1 : \llbracket \lambda \overline{x_1}. t_1 \rrbracket, \dots, P_n : \llbracket \lambda \overline{x_n}. t_n \rrbracket,$ $P_{\text{handle}} : I_{\text{handle}}, P_{\text{handle_special}} : I_{\text{handle_special}},$ $P_{\text{raise}} : I_{\text{raise}}, P_{\text{resume}} : I_{\text{resume}}$	$\llbracket v \rrbracket_\Gamma = \llbracket v \rrbracket_\Gamma^{\mathbf{r1}}$ $\llbracket v_0(v_1, \dots, v_n) \rrbracket_\Gamma =$ $\llbracket v_n \rrbracket_\Gamma^{\mathbf{rn}}; \dots; \llbracket v_0 \rrbracket_\Gamma^{\mathbf{r0}}; \text{call } \mathbf{r0}$ $\llbracket \text{handle } P_{\text{body}} \text{ with } P_{\text{op}} \text{ under } v_{\text{env}} \rrbracket_\Gamma =$ $\llbracket v_{\text{env}} \rrbracket_\Gamma^{\mathbf{r3}}; \text{mov } \mathbf{r2}, P_{\text{op}}; \text{mov } \mathbf{r1}, P_{\text{body}};$ $\text{call } P_{\text{handle}}$ $\llbracket \text{raise } v_1 v_2 \rrbracket_\Gamma =$ $\llbracket v_2 \rrbracket_\Gamma^{\mathbf{r2}}; \llbracket v_1 \rrbracket_\Gamma^{\mathbf{r1}}; \text{call } P_{\text{raise}}$ $\llbracket \text{resume } v_1 v_2 \rrbracket_\Gamma =$ $\llbracket v_2 \rrbracket_\Gamma^{\mathbf{r2}}; \llbracket v_1 \rrbracket_\Gamma^{\mathbf{r1}}; \text{call } P_{\text{resume}}$
$\llbracket \lambda \overline{x}. t \rrbracket = I$	
$\llbracket \lambda(x_1, \dots, x_n). t \rrbracket =$ $\text{push } \mathbf{rn}; \dots; \text{push } \mathbf{r1}; \llbracket t \rrbracket_{x_n, \dots, x_1};$ $\text{sfree } k; \text{return}$ where $k = n + t $	
$\llbracket t \rrbracket_\Gamma = I$	
$\llbracket \text{let } x = e \text{ in } t \rrbracket_\Gamma = \llbracket e \rrbracket_\Gamma; \text{push } \mathbf{r1}; \llbracket t \rrbracket_{\Gamma, x}$ $\llbracket v \text{ end} \rrbracket_\Gamma = \llbracket v \rrbracket_\Gamma^{\mathbf{r1}}$	$\llbracket v \rrbracket_\Gamma^{\mathbf{r}} = \iota$ $\llbracket x_i \rrbracket_{x_n, \dots, x_0}^{\mathbf{r}} = \text{load } \mathbf{r}, [\mathbf{sp} - i]$ $\llbracket i \rrbracket_\Gamma^{\mathbf{r}} = \text{mov } \mathbf{r}, i$ $\llbracket P \rrbracket_\Gamma^{\mathbf{r}} = \text{mov } \mathbf{r}, P$

Figure 2.11: Translation of selected LEXA constructs.

2.3.2 Translating handle, raise, and resume with Trampolines

Most of the work of effect handling happens in the built-in *trampoline* functions P_{handle} , P_{raise} , and P_{resume} , which are provided in Figure 2.12 as a reference.

To understand what the trampoline functions do, we illustrate concrete executions of the scheduler program (Figure 2.1) as control goes in and out of the trampolines. Figures 2.13 to 2.15 illustrate, respectively, the execution of a call to **handle**, **raise**, and **resume**. Each figure contains five state snapshots at different points during the execution of a trampoline: (1) before the trampoline is called, (2) at the beginning of the trampoline, (3) inside the trampoline, (4) before control is about to leave the trampoline, and (5) after control leaves the trampoline. Each snapshot consists of the register file and the heap-allocated stacks; registers and stacks unrelated to the discussion are omitted. Stack slots that belong to *header frames* (explained below) are highlighted in yellow. The use of $*$ as a location offset (e.g., P_{spawn}^*) means the exact offset is not relevant. It helps to pay

P_{handle} :	# r1 : P_{body} , r2 : P_{op} , r3 : L_{env}
0 mov r4 , sp ;	# save old stack pointer
1 mkstk sp ;	# create new stack and switch to it
2-5 push r2 ; push r3 ; push 0; push r4 ;	# create header frame; 0 is the general annotation (Section 2.4)
6 mov r5 , r1 ;	# r5 : P_{body}
7 mov r2 , sp ;	# r2 : exchanger slot
8 mov r1 , r3 ;	# r1 : L_{env}
9 call r5 ;	# call P_{body} with args in r1 , r2
10 pop r2 ;	# r2 : top of parent stack
11 sfree 3;	# deallocate remaining header frame
12 mov sp , r2 ;	# switch stacks
13 return	
P_{raise} :	# r1 : exchanger slot, r2 : op arg
0 load r4 , [r1];	# r4 : top of handler stack
1 store [r1], sp ;	# exchanger: top of resumption stack
2 mov sp , r4 ;	# switch stacks
3 load r5 , [r1 - 3];	# r5 : P_{op}
4-5 malloc r3 , 1; store [r3], r1 ;	# r3 : L_k
6 load r1 , [r1 - 2];	# r1 : L_{env}
7 jmp r5 ;	# call P_{op} with args in r1 , r2 , r3
P_{resume} :	# r1 : L_k , r2 : resumption arg
0 load r3 , [r1];	# r3 : exchanger slot
1 store [r1], ns ;	# invalidate resumption object
2 load r4 , [r3];	# r4 : top of resumption stack
3 store [r3], sp ;	# exchanger: top of handler stack
4 mov sp , r4 ;	# switch stacks
5 mov r1 , r2 ;	# r1 : resumption arg
6 return	

Figure 2.12: Built-in trampolines used by the translated code.

attention to the **ip** register, which points to the next instruction to execute.

Translating handle with P_{handle} . Figure 2.13 shows the state transitions as control enters the **handle** expression in **spawn** (line 19, Figure 2.1). (1) Control is inside P_{spawn} and about to enter the trampoline P_{handle} ; the registers store the arguments to P_{handle} . (2) Upon **call** P_{handle} , the return address is pushed onto the stack L_0 . (3) A new stack is allocated. Four values are pushed onto it: the address of the handler operation P_{yield} (P_{fork} is omitted in the presentation), the address of the closure environment L_{env} , the handler annotation **general** (Section 2.4 discusses annotations), and the old value of **sp**. We call this region of the stack the *header frame*, and we call its fourth slot—currently storing the old **sp**—the *exchanger*. As we will see, 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**. (4) Control is now about to transfer to P_{body} , the handled code, with two arguments in registers: the closure environment L_{env} and the location L_1^4 of the header frame. Here, the location of the header frame is used in P_{body} to identify the handler instance. (5) As control enters P_{body} , the return address and arguments are pushed onto the stack.

Translating raise with P_{raise} . Figure 2.14 shows the state transitions as an effect is raised (line 31, Figure 2.1), a resumption is captured, and control is transferred to a handler. (1) Control is inside P_{job} and about to enter P_{raise} : **r1** points to the header frame of the handler, and **r2** stores the operation argument. (2) Upon **call** P_{raise} , the return address is pushed onto the stack L_1 . (3) The contents of the exchanger and **sp** are swapped. (4) Control is about to leave P_{raise} for the handler code P_{yield} . A single-element tuple is allocated to store the address of the header frame; the address of this tuple, L_k , serves as the identity of the resumption and is placed in **r3** as the last argument to P_{yield} . (5) Control enters P_{yield} , and its arguments are pushed onto the stack L_0 . Notice that, importantly, no run-time search or stackwalking is needed to identify the handler or capture the resumption.

Translating resume with P_{resume} . Figure 2.15 shows the state transitions as a previously captured resumption is resumed (line 12, Figure 2.1). (1) Control is inside the caller of **resume** and about to enter P_{resume} : **r1** points to the resumption object, and **r2** stores the resumption argument. (2) Upon **call** P_{resume} , the return address is pushed onto the stack L_0 . (3) The pointer to the exchanger is loaded from the resumption object at L_k , the contents of the exchanger and **sp** are swapped, and the resumption object is updated to

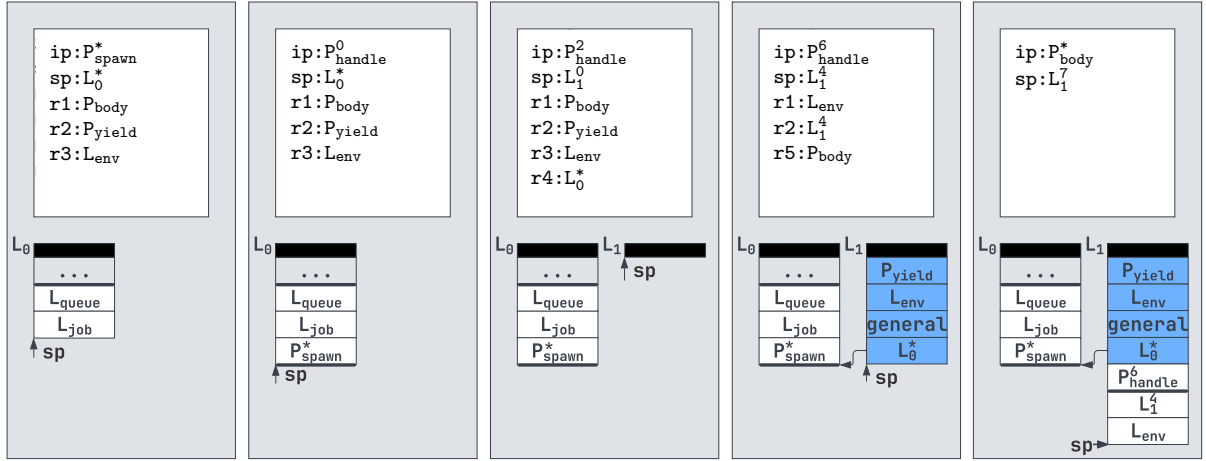


Figure 2.13: State transitions as control enters a **handle** expression and a handler instance is installed.

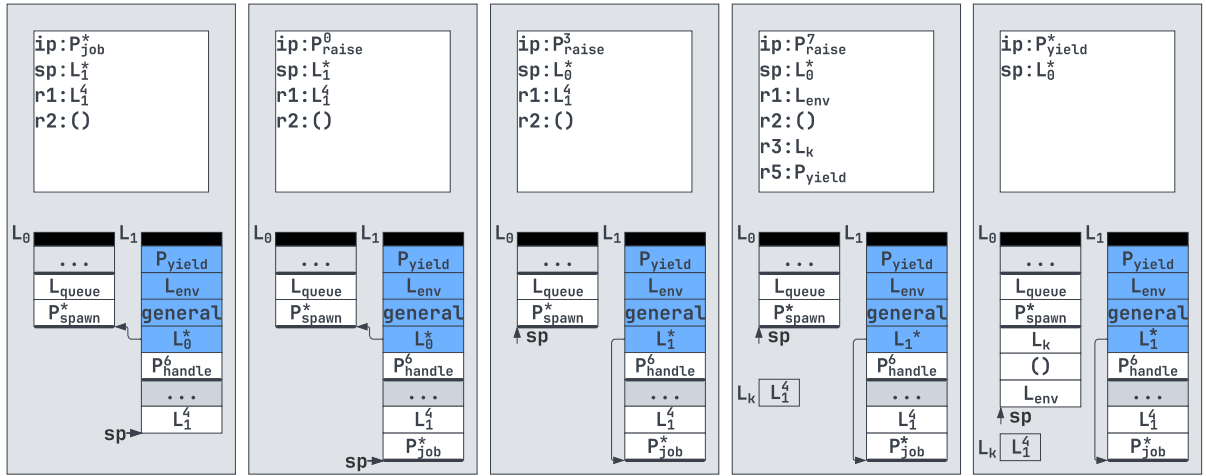


Figure 2.14: State transitions as an effect is raised, a resumption is captured, and control is transferred to a handler.

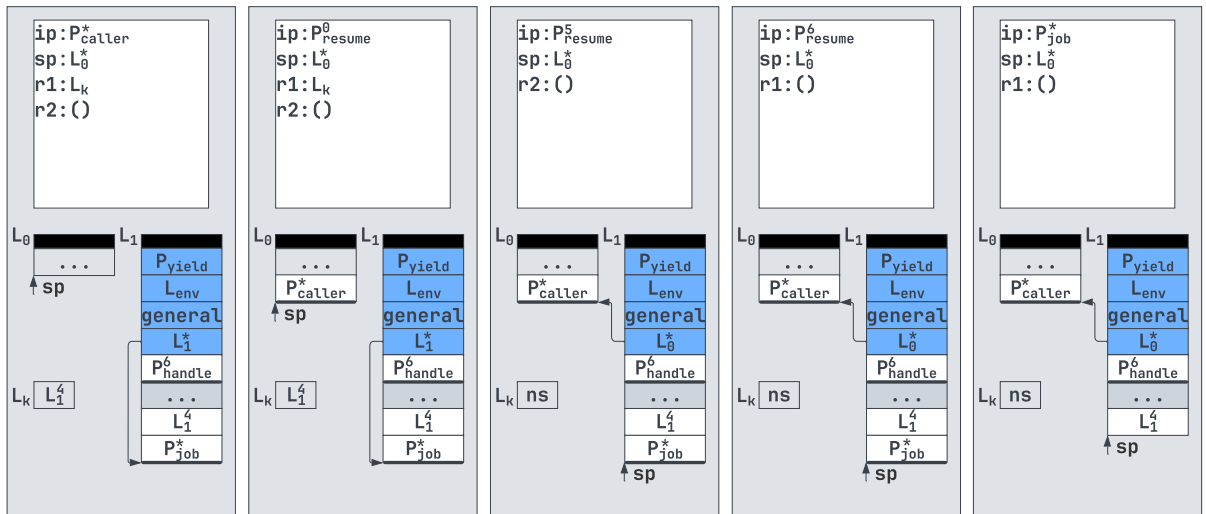


Figure 2.15: State transitions as a previously captured resumption is resumed.

prevent it from being resumed again. (4) Control is about to leave P_{resume} for the resumption (the previously suspended computation in P_{job}); the resumption argument is moved to **r1**. (5) A **return** instruction pops the return address off the stack L_1 and transfers control to the suspended computation in P_{job} .

2.4 Optimizing Tail-Resumptive and Abortive Handlers

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

Syntax. To capture the essence of how these special handlers are translated, we update the syntax of LEXA, as Figure 2.16 shows. **handle** expressions, as well as handler frames, are extended with annotations A that indicate whether the handler is tail-resumptive, abortive, or otherwise (a *general* handler). To simplify the formalism, **raise** expressions are also annotated, and the operational semantics requires that a handler handle only effects raised by a **raise** with the same annotation. Our compiler is more flexible: it does not require **raise** expressions to be annotated; rather, the **raise** implementation looks up the annotation in the handler’s header frame and dispatches accordingly.

Operational semantics. The **HANDLE** rule still works for all three kinds of handlers, needing only a metavariable A . The **RAISE** rule is updated to work only for **raise general**, and two additional rules are added for **raise tail** and **raise abort**, as Figure 2.16 shows. A **raise tail** expression is reduced *in place*, as if it were a regular function call. A **raise abort** expression is reduced by aborting the surrounding computation delimited by the handler frame. In both cases the handler implementation $M(P_{\text{op}})$ is no longer parameterized by a resumption k , and neither case reifies the resumption as a heap-allocated object, which justifies not allocating new stacks in the translation.

Importantly, even though tail-resumptive handlers are invoked in place, they can readily raise their own effects without any extra bookkeeping, thanks to the lexical scoping of

handler annotation $A ::= \text{tail} \mid \text{abort} \mid \text{general}$
 expression $e ::= \dots \mid \text{handle } P_{\text{body}} \text{ with } A \text{ } P_{\text{op}} \text{ under } v_{\text{env}} \mid \text{raise } A \text{ } v_1 \text{ } v_2$
 frame $F ::= \dots \mid \#L_A^{P_{\text{op}}, L_{\text{env}}} \square$

TAIL-RAISE

$$\begin{aligned}
 &\langle M \parallel H \parallel K \cdot (\#L_{\text{tail}}^{P_{\text{op}}, L_{\text{env}}} \square) \cdot K' \parallel E \parallel \text{let } x = \text{raise tail } v_1 \text{ } v_2 \text{ in } t \rangle \longrightarrow \\
 &\langle M \parallel H \parallel K \cdot (\#L_{\text{tail}}^{P_{\text{op}}, L_{\text{env}}} \square) \cdot K' \cdot (E, \text{let } x = \square \text{ in } t) \parallel [x_{\text{env}} \mapsto L_{\text{env}}, y \mapsto \hat{E}(v_2)] \parallel t' \rangle
 \end{aligned}$$

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

ABORT-RAISE

$$\begin{aligned}
 &\langle M \parallel H \parallel K \cdot (\#L_{\text{abort}}^{P_{\text{op}}, L_{\text{env}}} \square) \cdot K' \parallel E \parallel \text{let } x = \text{raise abort } v_1 \text{ } v_2 \text{ in } t \rangle \longrightarrow \\
 &\langle M \parallel H \parallel K \parallel [x_{\text{env}} \mapsto L_{\text{env}}, y \mapsto \hat{E}(v_2)] \parallel t' \rangle
 \end{aligned}$$

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

$\llbracket \text{general} \rrbracket^{\mathbf{r}} = \text{mov } \mathbf{r}, 0 \quad \llbracket \text{tail} \rrbracket^{\mathbf{r}} = \text{mov } \mathbf{r}, 1 \quad \llbracket \text{abort} \rrbracket^{\mathbf{r}} = \text{mov } \mathbf{r}, 2$

$\llbracket \text{handle } P_{\text{body}} \text{ with } A \text{ } P_{\text{op}} \text{ under } v_{\text{env}} \rrbracket_{\Gamma}$ $= \llbracket A \rrbracket^{\mathbf{r4}}_{\Gamma}; \llbracket v_{\text{env}} \rrbracket^{\mathbf{r3}}_{\Gamma};$ $\text{mov } \mathbf{r2}, P_{\text{op}};$ $\text{mov } \mathbf{r1}, P_{\text{body}};$ $\text{call } P_{\text{handle_special}}$ where $A = \text{tail}$ or $A = \text{abort}$	$\llbracket \text{raise tail } v_1 \text{ } v_2 \rrbracket_{\Gamma}$ $= \llbracket v_2 \rrbracket^{\mathbf{r2}}_{\Gamma}; \llbracket v_1 \rrbracket^{\mathbf{r1}}_{\Gamma};$ $\text{load } \mathbf{r3}, [\mathbf{r1} - 3];$ $\text{load } \mathbf{r1}, [\mathbf{r1} - 2];$ $\text{call } \mathbf{r3}$	$\llbracket \text{raise abort } v_1 \text{ } v_2 \rrbracket_{\Gamma}$ $= \llbracket v_2 \rrbracket^{\mathbf{r2}}_{\Gamma}; \llbracket v_1 \rrbracket^{\mathbf{r1}}_{\Gamma};$ $\text{load } \mathbf{r3}, [\mathbf{r1} - 3];$ $\text{mov } \mathbf{sp}, \mathbf{r1};$ $\text{load } \mathbf{r1}, [\mathbf{r1} - 2];$ $\text{sfree } 4; \text{ jmp } \mathbf{r3}$
--	---	--

$P_{\text{handle_special}} :$ $\text{push } \mathbf{r2}; \text{ push } \mathbf{r3}; \text{ push } \mathbf{r4}; \text{ push ns};$ $\text{mov } \mathbf{r5}, \mathbf{r1}; \text{ mov } \mathbf{r2}, \mathbf{sp}; \text{ mov } \mathbf{r1}, \mathbf{r3}; \text{ call } \mathbf{r5};$ $\text{sfree } 4; \text{ return}$	$\# \mathbf{r1}: P_{\text{body}}, \mathbf{r2}: P_{\text{op}}, \mathbf{r3}: L_{\text{env}}, \mathbf{r4}: A$ $\# \text{ create header frame on same stack}$ $\# \text{ call } P_{\text{body}} \text{ with args in } \mathbf{r1} \text{ and } \mathbf{r2}$ $\# \text{ deallocate header frame and return}$
---	--

Figure 2.16: Extending LEXA and the LEXA-to-SALT translation to address tail-resumptive and abortive handlers. The HANDLE and RAISE rules are omitted as they require minimal changes to the rules in Figure 2.5.

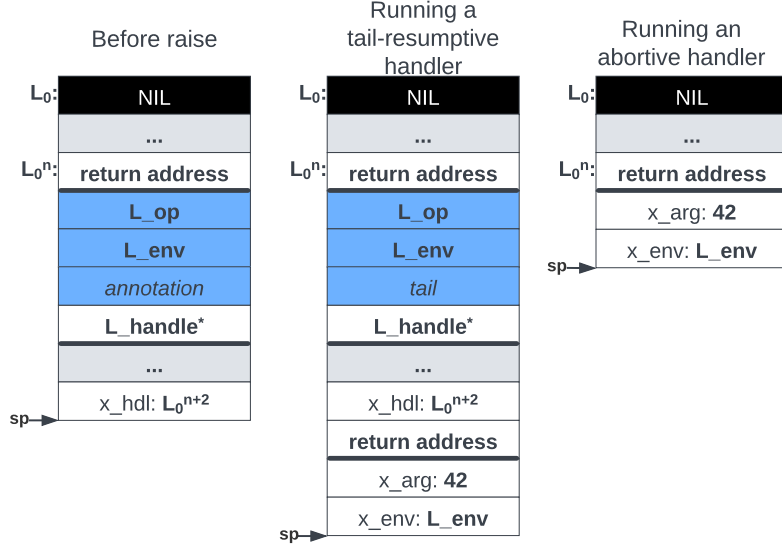


Figure 2.17: Stack layouts for tail-resumptive and abortive handlers, which place the header frame on the parent’s stack rather than on a freshly allocated stack.

handlers. In languages where handlers are dynamically scoped, in-place invocation of tail-resumptive handlers is tricky to get right: extra care must be taken to ensure that the in-place invocation does not bring unintended handlers into the dynamic context. For example, in the work of Xie and Leijen [66], the tail-resume optimization requires a special **under** construct that filters out handlers that should be skipped when the tail-resumptive handler raises its own effects. In contrast, with lexical scoping, a tail-resumptive handler has already captured the right handlers in its lexical closure, so it can be invoked directly without extra machinery. This simplicity speaks to the well-behaved nature of the lexical-scoping semantics.

Translation. Figure 2.16 also shows that the translation of **handle** is specialized for tail-resumptive and abortive handlers: it allocates the header frame on the same stack as the parent, using the trampoline $P_{\text{handle.special}}$ instead of P_{handle} . Notice that $P_{\text{handle.special}}$ uses no **mkstk** instruction, and that it pushes an **ns** value in place of the exchanger, which is needed only when a resumption has to be reified and stacks switched. The translation of **raise tail** looks like a regular function call. The translation of **raise abort** cuts the stack to the abortive handler’s header frame and then jumps to the handler’s code.

Figure 2.17 illustrates the resulting stack layouts. Because the header frame sits on the parent stack and no new stack is allocated, a tail-resumptive raise is essentially a function call, and an abortive raise is essentially a stack cut followed by a jump. As we discuss in Section 2.6, these special cases also avoid all heap allocation, and—because no assembly-level stack switching is involved for tail-resumptive handlers—they expose the generated code to the full force of downstream optimization.

2.5 Correctness of the Translation

To show that the translation is semantics-preserving, we give a simulation proof relating the execution of a program in LEXA to the execution of the translated program in SALT. Our theoretical framework strictly follows Leroy [26].

We first define *initial* and *final* configuration predicates for both languages. The initial configuration of LEXA loads the top-level functions into the code memory, together with an additional P_{init} that calls the main function and exits; the heap, evaluation context, and local environment are empty. The initial configuration of SALT loads the translated instruction sequences, together with a P_{init} that calls the main function and halts; the heap contains an initial empty stack, **sp** points to it, and **ip** points to P_{init} . In a final configuration of LEXA, the term is **halt** and the value in the current local environment is the result; in a final configuration of SALT, **ip** points to a **halt** instruction and the value at the top of the active stack is the result.

Starting from an initial configuration, a program that reaches a final configuration with result i after finitely many steps is said to converge(i); a program that reaches a non-final configuration from which no step is possible is stuck; and a program that steps forever diverges. These are the *observable behaviours* of a program.

Definition 2.1 (Semantic preservation). *A translation function $\llbracket \cdot \rrbracket$ is semantics-preserving if, whenever $G \Downarrow B$ and $B \neq \text{stuck}$, we have $\llbracket G \rrbracket \Downarrow B$.*

To prove that our translation preserves semantics, we use a simulation argument: we construct a relation $\sim$ relating LEXA configurations to SALT configurations and show that it is preserved by the reduction relations of both languages.

Definition 2.2 (Simulation). *A target-language reduction relation $\rightarrow$ simulates a source-language reduction relation $\rightarrow$ with respect to a relation $\sim$ if, for all C_1 , C_2 , and C'_1 such that $C_1 \sim C_2$ and $C_1 \rightarrow C'_1$, there exists C'_2 such that $C_2 \rightarrow^+ C'_2$ and $C'_1 \sim C'_2$.*

Theorem 2.1. *A translation $\llbracket \cdot \rrbracket$ from a source language $\mathcal{L}_1$ to a target language $\mathcal{L}_2$ is semantics-preserving if, for every $\mathcal{L}_1$ program G_1 and its translation $G_2 = \llbracket G_1 \rrbracket$, the following hold:*

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

The proof of Theorem 2.1 can be found in Leroy [26]. Given the theorem, all that remains is to construct the simulation relation $\sim$ and show that it satisfies the three conditions. The relation is defined using several auxiliary relations that relate components of the configurations. Some are indexed by a context Ξ that maps LEXA labels to SALT addresses; the mapping for most labels can be statically determined, except for the handler labels of tail-resumptive and abortive handlers, which correspond to addresses in the middle of a stack whose offsets from the base are known only at run time. The most interesting auxiliary relation connects the current point of control in the two languages: it relates a LEXA evaluation context to a SALT heap, where one LEXA evaluation context can correspond to multiple SALT stacks (one per general handler installed), and it relates the most recently pushed stack frame to the location pointed to by **sp**. A LEXA resumption, in turn, corresponds to a single-element tuple and a collection of stacks in SALT.

Theorem 2.2. *The translation from LEXA to SALT is semantics-preserving.*

Theorem 2.2 is reduced to proving that the simulation relation $\sim$ satisfies the three conditions of Theorem 2.1. The proof is largely mechanical; the full development, including the auxiliary relations and the case analysis, can be found in Appendix A. We have also implemented the formalism faithfully in Racket, including the translation and an interpreter for the target language.

2.6 The LEXA Compiler

The previous sections provide a formal account of the compilation process. This section describes the implementation of our compiler for LEXA. Unlike the formalization, which assumes programs have undergone closure conversion, the compiler supports a friendlier surface syntax for LEXA. It applies closure conversion and hoisting before compiling to C; the compiled C program is then fed to LLVM for optimization and code generation, and

the generated code is linked with the Boehm–Demers–Weiser garbage collector [5]. The compilation from LEXA to C is syntax-directed and does no optimization of its own; we leave all standard optimizations to LLVM.

At the heart of the back end is a C library, called StackTrek, that provides low-level facilities for stack switching. Its design follows the formalization closely. StackTrek consists of x86-64 assembly functions responsible for stack switching, together with a set of user-facing macros that are useful in themselves to a C programmer who wants to write programs with lexical effect handlers.

2.6.1 Overview

StackTrek provides three macros for stack switching: `HANDLE`, `RAISE`, and `RESUME`. They correspond to the built-in functions P_{handle} , P_{raise} , and P_{resume} in the formal translation (Section 2.3). Figure 2.18 shows how the macros are used in compiled C code for the scheduler example of Figure 2.1. To improve readability we use C global variables and remove the closure environment parameters, and we omit the `Tick` and `Exn` handlers; an actual program would need to handle the exceptional case of an empty queue at line 5.

`HANDLE` (line 20) takes the code to be handled and the handler, which contains an implementation and an annotation for each effect operation. 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 handled code. Although `HANDLE` contains several branches for handlers of different annotations, the branch to take is always resolved at compile time, so `HANDLE` is essentially executed as a linear sequence of instructions. `RAISE` (lines 29 and 34) takes a pointer to the header frame identifying the handler, the operation index, and the arguments; it invokes the operation implementation and, depending on the annotation stored in the header frame, may switch stacks. `RESUME` (line 6) takes the resumption and the value to resume it with.

2.6.2 Limited Support for Multishot Resumptions

A *multishot* resumption is one that is resumed more than once. Our formal translation rules this out: it enforces at run time that each resumption is resumed at most once, the same restriction adopted by Multicore OCaml [48]. Our implementation goes further and does support multishot resumptions, but only under the conditions we make precise below.

Recall that the compiler gives each handler a run-time identity, the address of its header frame, and that a raise site reaches its handler through this address (Section 2.3).

```

1 queue_t* q = queueMake();
2 int n_jobs = 1000;
3
4 void driver() {
5     resumption_t* k = queueDeq(q);
6     RESUME(k, 0);
7     driver();
8 }
9
10 void yield(resumption_t* k) {
11     queueEnq(q, k);
12 }
13
14 void fork(void* g, resumption_t* k) {
15     queueEnq(q, k);
16     spawn(g);
17 }
18
19 void spawn(void (*f)(header_t*)) {
20     HANDLE(f, ({GENERAL, yield}, {GENERAL, fork}));
21 }
22
23 void scheduler(void (*f)(header_t*)) {
24     spawn(f);
25     driver();
26 }
27
28 void job(header_t* P) {
29     ... RAISE(P, 0, ()); ...
30 }
31
32 void jobs(header_t* P, header_t* T) {
33     for (int i = n_jobs; i > 0; i--)
34         RAISE(P, 1, (job));
35 }
36
37 int main() {
38     init_stack_pool();
39     scheduler(jobs);
40     destroy_stack_pool();
41 }

```

Figure 2.18: The scheduler example in C using the StackTrek macros.

A captured resumption therefore holds, within its stacks, references to the header frames of its handlers. Resuming such a resumption more than once requires *copying* its stacks, since the original must be preserved for any later resumption. Herein lies the difficulty: when the stacks are copied, the references they contain are copied verbatim, so they still point to the header frames on the *original* stacks, whereas we want them to point to the corresponding header frames on the *copied* stacks. Without fixing up the references or redirecting them, the copies of the resumption effectively share the same header frame, and they might corrupt the exchanger slot that stores the parent stack information of another copy.

That said, our implementation still supports multishot resumptions under certain restrictions: a multishot resumption must span no more than one stack, and at most one copy of it may be installed on the active stack at any time. Under these conditions the header frames can safely be shared among the copies, because the exchanger slot will not be inadvertently corrupted. These restrictions are mild in practice: every benchmark from prior work [38] and from our own evaluation (Section 2.7) can still be expressed idiomatically. Chapter 3 introduces a redirection technique that lifts the restrictions altogether.

2.6.3 Stack-Switching Functions and Calling Convention

At the core of each StackTrek macro is a call to a function containing inline assembly; we call these *stack-switching functions*. They are written as **naked** C functions so the compiler generates no prologue or epilogue. Wrapping the assembly in C functions is convenient because the C compiler respects calling-convention annotations and thus saves and restores register state across the call and return. Figure 2.19 shows the stack-switching function used by the RAISE macro.

In SALT, registers serve only to shuttle values between memory and the processor. A real machine, however, also uses registers to hold program state. The System V calling convention for x86-64 prescribes that about half of the registers be callee-saved and the other half caller-saved. This works when control flow is local, but in the presence of stack switching a caller might be returned to from a different callee that does not hold the register state the caller expects. A common workaround is to use `setjmp` and `longjmp` to save and restore register state as part of stack switching.

The OCaml compiler takes a different approach: it uses a calling convention where all registers are caller-saved. This makes stack switching cheap, because it does not matter if control returns from a different callee—the caller restores the register state by itself. We refer to this convention as **preserve_none**. The convention is not without cost: it

```

1  __attribute__((naked, preserve_none))
2  int64_t save_switch_and_run_handler(
3      intptr_t* env, int64_t arg, void* exec, void* func) {
4      __asm__ (
5          "movq 0(%%rdx), %%rax" // Load sp from the exchanger
6          "movq %%rsp, 0(%%rdx)" // Save sp to the exchanger
7          "movq %%rax, %%rsp"   // Switch to the new stack
8          "jmpq *%%rcx"         // Call the handler; the first three
9                               // args are in the right registers
10     );}

```

Figure 2.19: Stack-switching function for the RAISE macro.

may save and restore registers across calls that do not actually use them. In our implementation we therefore use a hybrid convention: System V for most user functions, and `preserve_none` for the stack-switching functions and for functions that implement effect operations. The `preserve_none` attribute serves as a hint to LLVM. This choice simplifies StackTrek, because the stack-switching functions are no longer responsible for saving and restoring callee-saved registers via `setjmp/longjmp`; it also facilitates optimizations, as we discuss next, while preserving the usual System V performance profile for most user functions.

Lifting the overhead out of the hot path. When an effect operation is invoked in a hot path, the cost of saving and restoring callee-saved registers can be significant. Consider line 34 of Figure 2.18, where `fork` is invoked in a tight loop. Under System V, the callee-saved registers would have to be saved inside the stack-switching function every time `fork` is raised, a major cost in this tight loop. Under `preserve_none`, since there are no callee-saved registers, the stack-switching function does not save and restore any registers, so the loop is lightweight. However, now all registers are caller-saved, which seems to just move the cost to the caller. Luckily, the compiler is able to move the saving and restoring to the prologue and epilogue of the caller (`jobs`), doing it only once.

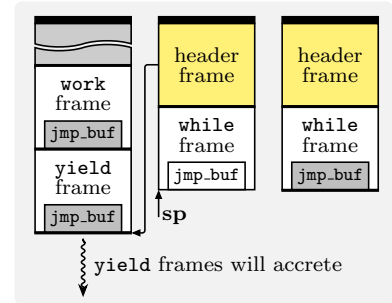
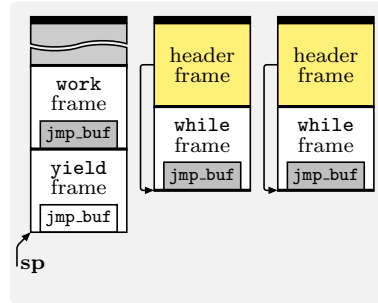
Exposing tail-call optimization opportunities. The `preserve_none` convention also enables tail-call optimizations that would be missed if `setjmp` and `longjmp` were used for stack switching. Figure 2.20a shows a round-robin scheduler between two tasks: when the running task yields (line 5), the parked task is resumed (line 13) in a tail position of the

```

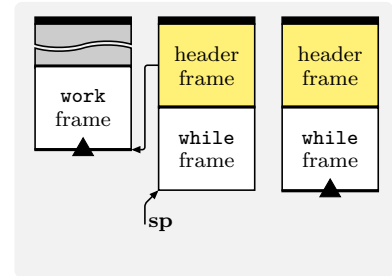
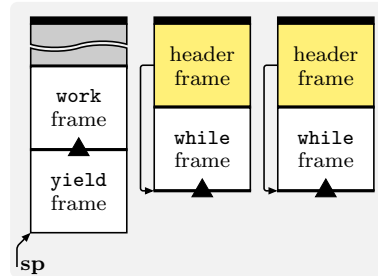
1 val parked = ref None;
2 val work = fun() {
3   handle
4     while true
5       raise P.yield ()
6     done
7   with P : Process =
8     | yield _ k =>
9       val peer = !parked
10      parked := Some k
11      match peer with
12      | Some k' =>
13        resume k' ()
14      | None => ()
15   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

```

(a) Round-robin scheduling with effect handlers.



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



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

Figure 2.20: For the program in (a), using `setjmp` and `longjmp` for stack switching can lead to stack overflow (b), while the `preserve_none` calling convention used by the LEXA compiler allows tail calls to be optimized (c).

`yield` handler.² This scheduler resembles the one used by Multicore OCaml [48], in which a paused computation is resumed in a tail position of a handler.

If stack switching were implemented with `setjmp` and `longjmp`, each switch would save the callee-saved registers on the current stack in a `jmp_buf`, so the `yield` frames would accumulate and the stack would grow with every `yield`, eventually overflowing

²This handler for `yield` is not tail-resumptive, however. Although `k'` is resumed in a tail position of the handler, it is not the same as the current handler's resumption `k`, so the tail-resume optimization does not apply.

(Figure 2.20b). The `preserve_none` convention avoids this: the caller of a stack-switching function saves its register state before the call, so the boundary between caller and callee—the black triangles in Figure 2.20c—by itself records the information needed to resume the paused computation, and any code holding a stack pointer to such a boundary can resume the computation simply by executing a `ret`, just as in our formalization. Because the `RESUME` call that `resume k' ()` (line 13) compiles to is in tail position, the compiler pops the `yield` frame before the call, and because the stack-switching function itself has no frame, the stack size stays constant.³

2.6.4 Reducing Allocations

For efficiency, StackTrek aims to reduce heap allocations. For stacks, we use fixed-size *stack segments*, recycled through a stack pool, following prior work on implementing stacks and continuations [16, 48]. The pool is a contiguous block of memory divided into fixed-size stack segments, with a global bitmap tracking availability; when the runtime needs more stack segments than are available, it allocates a new block on the spot. Stack segments are reclaimed by a combination of manual memory management and garbage collection: StackTrek releases a stack segment back to the pool when it is certain the segment is no longer needed—for instance, when the body of a `handle` expression completes, or when an abortive handler is invoked and control transfers to an earlier stack segment. A special macro `RESUME_FINAL` indicates that a resumption is being resumed for the last time, so that no copy of the stack segment is made and the segment is released afterwards. Stack segments not released manually are reclaimed by the garbage collector. For header frames, StackTrek saves an allocation by allocating the handler’s closure environment directly as part of the header frame; consequently, tail-resumptive and abortive handlers incur no heap allocation at all.

2.7 Evaluation

We evaluate our implementation of LEXA against other implementations of effect handlers, on a benchmark suite maintained by the community [15]. We add three benchmarks: `Resume Nontail 2`, `Scheduler`, and `Interruptible Iterator`. The first tests the efficiency of capturing and resuming deep resumption stacks; the other two represent real-world uses of

³This tail-call optimization requires that the caller `yield` also use the `preserve_none` calling convention, which is the case in our implementation.

effect handlers and test the efficiency of effect propagation in the presence of deep stacks of handlers. The experiments were conducted on a workstation with a 3.5 GHz CPU.

We compare LEXA with the two other languages that support lexical effect handlers, Effekt and Koka, and we use as baselines two languages that support dynamically scoped handlers, Koka and OCaml. Koka is best known for its dynamically scoped handlers but has recently incorporated *named handlers* [65], a form of lexical handlers. We use Effekt 0.2.2, Koka 3.1.1, and OCaml 5.3.0, and for OCaml the multicont library [22] for multishot resumptions. For most benchmarks we use Effekt’s MLton back end, as it produces the fastest code; for the benchmarks it cannot handle, we resort to the fastest back ends that can: Chez Scheme for **Generator** and **Handler Sieve**, and Node.js for **Scheduler** and **Interruptible Iterator**. (These benchmarks feature recursive, handler-polymorphic functions, which Effekt CPS-translates into stack-shape-polymorphic functions that MLton’s monomorphization requirement rules out.) The systems differ in many ways besides their handling of effects—most notably, LEXA has only limited support for multishot resumptions, whereas the others support them without restriction; LEXA, Effekt’s MLton back end, and OCaml use garbage collectors, whereas Koka uses reference counting; and Koka uses GCC as its back end whereas LEXA uses LLVM. We advise the reader to take this potential for bias into account.

Results. The results are presented in Table 2.1 and Figure 2.21. LEXA is the fastest system on eight of the 14 benchmarks; on the rest it is the second fastest. Figure 2.21 shows scaling plots, with input size on the x -axis and running time on the y -axis; LEXA is the most scalable system on the same eight benchmarks. Compared to Effekt, LEXA fares particularly well on **Handler Sieve**, **Scheduler**, and **Interruptible Iterator**, which involve recursion and thus lead to deep stacks of handlers at run time. Also notable are **Countdown** and **Iterator**, where LEXA compiles the program to a constant: most credit belongs to LLVM, but that LEXA generates code amenable to such optimizations is a testament to its quality—in particular, because tail-resumptive handlers allocate the header frame on the same stack, no assembly is involved, allowing LLVM to optimize without hindrance.

LEXA is less efficient on benchmarks that use multishot resumptions (**Nqueens**, **Tree Explore**, and **Triples**), where Effekt or OCaml is the fastest; Effekt represents resumptions as immutable closures and so need not copy them when they are used more than once. The **Resume Nontail 2** benchmark forces the stack to grow with the input size and thus stresses the cost of capturing and resuming deep resumption stacks; both LEXA and Effekt scale linearly, while Koka (regular and named) scales super-linearly, which suggests that capturing a resumption is not a constant-time operation in Koka—Koka builds up the

Table 2.1: Benchmark running times measured for different systems. “n/a” indicates that the benchmark could not be implemented in the respective system. For benchmarks that we could not compile with Effekt’s MLton back end, we use the fastest back ends that can handle them (Chez Scheme or Node.js). The fastest result for each benchmark is highlighted.

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

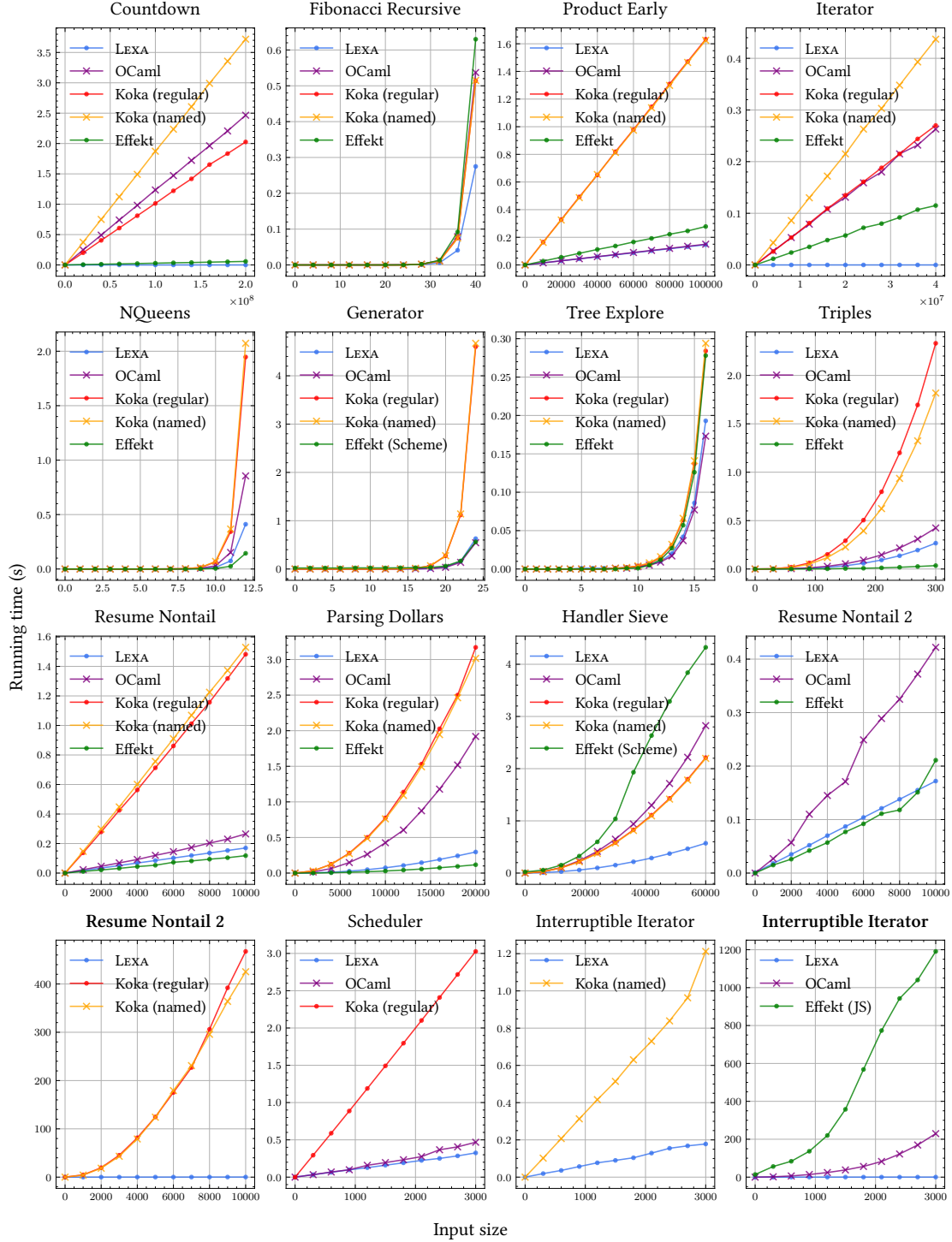


Figure 2.21: Scaling behaviours of different systems on 14 benchmarks. Effekt is omitted from the Scheduler chart; it appears in Figure 2.2. Resume Nontail 2 and Interruptible Iterator are each represented by two charts; the second chart (with the title in bold) reports on the systems that exhibit super-linear scaling, using LEXA as a reference point.

resumption frame by frame, which takes longer as the stack grows. OCaml is piecewise linear on this benchmark, likely a consequence of its stack management strategy.

The **Scheduler** benchmark, featuring **Tick** effects, is the program of Figure 2.1. Effekt does not scale on it, as discussed in Section 2.1. Interestingly, OCaml scales well: its exception handling is implemented differently from effect handling, by passing the code address and stack location of an exception handler to the raise site. Koka (regular) also scales linearly, but for a different reason—tail resumption: because the **Tick** handler is tail-resumptive (line 50) and Koka performs the tail-resume optimization, **Tick** is invoked in place without reifying a resumption; removing the tail-resume hint degrades Koka to super-linear scaling. The **Interruptible Iterator** benchmark, adapted from prior work [30, 70], uses bidirectional effects [70] to let the client of a coroutine iterator concurrently update a list during iteration. Like **Scheduler**, it leads to $O(n)$ recursion depth and installs $O(n)$ handlers, but unlike **Scheduler** the installed handlers cannot be avoided by adjusting lexical scopes. Effekt and OCaml both scale super-linearly on it, while Koka with named handlers scales linearly—again degrading to super-linear scaling when the tail-resume hint is removed.

2.8 Related Work

Zhang et al. [69] demonstrate that dynamically scoped exception handlers can lead to exceptions being caught by the wrong handler. To address this, they introduce *tunneled exceptions* and a type system enforcing that exceptions tunnel through program contexts oblivious to them. Tunneling is essentially a form of lexically scoped exception handling. They implement tunneled exceptions for the Genus programming language [67], compiling them to unchecked Java exceptions: installing a handler generates a fresh identifier that is passed down the call chain as a capability, and upon an exception the call stack is walked to find a handler with the matching identifier. Zhang and Myers [68] suggest that the loss of local reasoning with dynamically scoped handlers can be analogized to a loss of *parametricity* [59]. They prove, for a type-and-effect system supporting *effect polymorphism*, that local-reasoning principles are restored by lexical effect handlers, using a logical-relations argument.

Biernacki et al. [4] coin the term *lexically scoped handlers* and study two semantics for them, *open* and *generative*, showing that generativity is necessary when effect operations can be polymorphic. Existing languages supporting lexical handlers, including LEXA, freshly generate labels for each installed handler instance at run time. This generativity is also seen with multi-prompt delimited control [20]. Indeed, StackTrek is similar to libmprompt [25], a C/C++ library for multi-prompt delimited control: both support

constant-time stack switching, but StackTrek additionally avoids allocating new stacks when the delimited continuation is not used or is invoked in a tail position.

Brachthäuser et al. [7] present Effekt’s type-and-effect system for lexically scoped handlers, featuring lightweight effect polymorphism via second-class values, and translate it to System Ξ , a calculus where handlers are passed explicitly. System Ξ is then translated to a region-based calculus Λ_{cap} via lift inference [38], which is further compiled to System F [45]. System Ξ is similar to LEXA in that both are intermediate languages where handlers are passed explicitly; it differs in being typed and in imposing a second-class restriction on functions, handlers, and continuations, which simplifies lift inference. System C [6] extends System Ξ with first-class functions, but how to translate System C to Λ_{cap} is left as future work. Besides lift inference, Effekt has back ends that target a monad for multi-prompt delimited control [14, 8]. As discussed in Section 2.1, both strategies involve run-time computations to lift a raised effect out of the extents of its dynamically enclosing handlers until the right handler is found.

WasmFX is a proposal for adding effect handlers to WebAssembly [40]. It is largely based on dynamically scoped handlers but does consider *named handlers*, a form of lexical handlers given a generative semantics. Xie et al. [65] present a type-and-effect system for first-class named handlers in Koka. Koka was initially designed around dynamically scoped handlers [24], and Xie and Leijen [66] formalize the compilation of dynamically scoped handlers to a language with multi-prompt delimited control, passing *evidence vectors* that can be looked up to find the correct handler; named handlers in Koka are implemented on the same mechanism, passing evidence as first-class values. The LEXA compiler instead targets a lower-level language, manipulating stack layout directly.

Liu et al. [30] introduce *interruptible iterators* in the JMatch language, with two back ends: a CPS translation to Java and a direct translation to C++. Zhang et al. [70] introduce *bidirectional effects*, subsuming interruptible iterators and generalizing formalisms of effect handlers, with applications such as `async-await` with exceptions; a semantic soundness result is established, while an implementation is left as future work. Our formalization and implementation of LEXA support bidirectional effects. It is also possible to encode bidirectional effects by passing thunked effectful computations to resumptions [70].

2.9 Summary

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

of a run-time search for handlers, and it is guided by a formal model of the translation that is proven semantics-preserving. The upshot is that the lexical-scoping semantics of effect handlers not only affords local reasoning principles, as previously established, but also enables good performance, as our empirical results suggest. The latest release of the LEXA compiler is available at <https://github.com/lexa-lang/lexa>. The next two chapters build on this foundation: Chapter 3 extends the direct approach to full multishot resumptions, and Chapter 4 restores the zero-overhead principle for exceptional effects.

Chapter 3

Zone LEXA: Virtualizing Continuations

Chapter 2 established a stack-switching implementation of lexical effect handlers in which the stack address of a handler serves as its run-time identity, so that raising an effect needs no run-time search: the runtime jumps directly to the handler whose address was passed down to the raise site. That technique, however, breaks down in the presence of *multishot* resumptions. Resuming a continuation more than once requires copying its stack segments, and the copied segments retain handler references that still point into the *original* stack. This chapter presents *Zone LEXA*, an implementation strategy that fully supports multishot resumptions—including running multiple copies of the same resumption in parallel—while preserving the constant-time handler lookup that makes a stack-switching runtime fast. The central idea is to make handler references *virtual* and to translate them to physical stack addresses on demand, through a mechanism inspired by virtual memory in operating systems. This chapter is based on “Virtualizing Continuations” [35], published at PLDI 2026.

3.1 The Multishot Challenge

Raising an effect suspends the current computation and captures the delimited continuation (a.k.a. the *resumption*) up to the corresponding handler. The handler may then choose whether, when, and how many times to resume the continuation. The implementation techniques studied so far in this thesis, based on stack switching, use *stack addresses* as

unique identifiers for the handlers installed on the stack. This approach avoids searching the stack for the appropriate handler when an effect is raised: the runtime can directly locate the handler via its stack address, thereby enabling efficient handler lookup and continuation capture. The technique was first introduced in LEXA [33] (Chapter 2) and subsequently adopted by Effekt [37]. Unfortunately, this implementation strategy appears incompatible with multishot continuations.

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

For example, Figure 3.1 shows how a simplified version of depth-limited expectimax search [43] can be implemented with the help of lexical effect handlers and multishot resumptions. Lines 7–9 give a generic framework for defining stochastic search problems, and lines 14–25 implement the expectimax search algorithm.

The `step` function models a single step of the agent’s interaction with the environment: the agent chooses an action, which causes the environment to stochastically transition to a new state. The state transition function `nextState` is probabilistic; hence it needs the capability to perform the `Probability` effect. The `expectimax` function computes the *value* of a state `s` (i.e., the optimal expected utility) with a planning horizon of `n` steps, by recursively computing the value of the next state with a horizon of `n - 1` steps.

Raising an effect (in `step` and in `nextState`) transfers control to the handler, while also capturing the delimited continuation up to the handler. The handlers, defined on lines 20–25, interpret the `Decision` and `Probability` effects by exploring all possible choices of actions and all possible state transitions. The handler for `Decision` interprets `choose` by invoking the resumption `k` with every possible choice of action—multishot resumptions!—and taking the maximal value. The handler for `Probability` interprets `sample` by invoking the resumption `k` with every possible state from the distribution’s support—again, multishot resumptions!—and computing the expected value averaged over all possible state transitions. By allowing the continuation to be resumed multiple times, multishot resumptions provide an elegant way to travel back in time and explore more than one possible future.

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

```

1  effect Decision =
2  | choose (actions: list[Action]) : Action
3  type Dist['a] = list[(float, 'a)]
4  effect Probability =
5  | sample (dist: Dist['a]) : 'a
6
7  def actions (s: State) : list[Action]
8  def nextState (p: Probability) (s: State) (a: Action) : State
9  def utility (s: State) : float
10
11 def step (d: Decision) (p: Probability) (s: State) : State =
12   let a = raise d.choose (actions s) in nextState p s a
13
14 def expectimax (n: int) (s: State) : float =
15   if n == 0 then utility s
16   else
17     handle (d: Decision) (p: Probability) =>
18       expectimax (n - 1) (step d p s)
19     with
20       | choose choices k =>
21         List.max
22           (List.map choices (fun c => resume k c))
23       | sample dist k =>
24         List.sum
25           (List.map dist (fun (prob, s) => prob * resume k s))

```

Figure 3.1: Defining and solving stochastic search problems. The effect handlers use multishot resumptions.

```

1      | choose choices k =>
2          List.parMax
3          (List.parMap choices (fun c => resume k c))
4      | sample dist k =>
5          List.parSum
6          (List.parMap dist (fun (prob, s) => prob * resume k s))

```

Figure 3.2: A parallelized version (simplified) of the handler code for the **choose** and **sample** operations. This code leads to run-time errors in both Base LEXA and Effekt.

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

This situation arises in LEXA and Effekt, as they use stack addresses to identify handlers installed on the stack. After copying, the handler references in the copied stack still point to handlers on the original stack.¹ Continuing to use these stale references causes incorrect behaviour and potentially security vulnerabilities. It is possible to work around this problem by scanning the stack and fixing up all stale references, but this approach is known to be costly [53, 54].

As a result, Base LEXA outright forbids stacks captured for multishot resumptions from containing handlers; it aborts the program if such a situation is detected at run time [33]. For this reason, the code in Figure 3.1 would cause a run-time error in Base LEXA. Effekt lifts this restriction by introducing a global pointer that routes handler references to the active resumption copy. However, this approach still disallows a captured continuation from being resumed while another copy of it is active; Effekt aborts the program if such a situation is detected at run time [37].

Importantly, this latter restriction, shared by both LEXA and Effekt, prevents parallel execution of multishot resumptions. We make an observation that is perhaps obvious but often overlooked in current research on effect handlers: many applications of multishot

¹This problem is not unique to lexical effect handlers. It also arises in other settings that involve stack copying and *stack-allocated* (i.e., lexical) resources, since stack allocation leads to pointers into the stack. We focus on lexical effect handlers, but the ideas presented here can be adapted to other settings as well. In fact, our implementation already supports a form of stack-allocated state using the same technique (Section 3.6.4).

resumptions are *embarrassingly parallel*, meaning that individual resumptions can be executed independently with little communication. For example, in Figure 3.1, the handling of both the nondeterministic choices of actions and the probabilistic outcomes of state transitions ought to be easily parallelizable: Figure 3.2 presents a parallelized version of the handler code, where different copies of the same resumption are run in parallel using `List.parMap`. This parallelization could speed up expectimax search on modern multicore processors. Unfortunately, the restrictions in LEXA and Effekt make them incompatible with such simple forms of parallelism, limiting the usefulness of multishot resumptions in practice.

Our approach. This chapter presents a novel implementation strategy for lexical effect handlers that fully supports multishot resumptions, by virtualizing resumptions. We draw an analogy between managing multishot resumptions in a language runtime and managing multiple processes in an operating system (OS). An OS uses virtual memory to provide each process with a virtual view of its address space. Similarly, we provide each copy of a resumption with a virtual view of the identities of the handlers installed within it, so that code running in the resumption copy need not care where stack copying has relocated the handlers, even when multiple copies of the same resumption are run in parallel.²

While modern OSes rely on hardware memory-management units to perform address translation, our approach uses a *software*-based memory-management unit (sMMU) to translate virtual handler references to concrete stack addresses that uniquely identify handlers on the stack. A complicating difference from traditional OS virtual memory is that, while each OS process can access only its own address space, a resumption may access handlers on enclosing resumptions’ stacks. Fortunately, the lexical scoping discipline of handlers enables static reasoning about which handlers are accessible to a resumption. This observation lets us design an efficient address-translation mechanism that walks a linked list of *zones*—rather than finer-grained stack frames or stack segments—thereby minimizing the run-time work needed for handler lookup.

The chapter is organized as follows. Sections 3.2 and 3.3 present the main ideas from a language implementer’s perspective. Section 3.4 formalizes the new implementation strategy as an operational semantics for lexical effect handlers, and Section 3.5 proves that the new semantics is correct with respect to the standard semantics. Section 3.6 gives a more detailed view of the implementation. The evaluation in Section 3.7 suggests that

²We do not take the analogy too far. It pertains only to address-space abstraction and the idea of address translation; the actual mechanisms for address translation differ. We do not claim analogies to other aspects of virtual memory, such as paging.

our approach enables significant speedups for embarrassingly parallel multishot resumptions while incurring minimal overhead on zero- and single-shot resumptions. Section 3.8 discusses related work, and Section 3.9 concludes.

3.2 Background: Handler Lookup in a Stack-Switching Runtime

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

Therefore, the compiler must efficiently implement two operations: (1) finding the appropriate handler for a given effect, and (2) capturing, and later resuming, the continuation between the raise site and the handler. The second operation is easy to implement efficiently: once the correct handler is located, capturing the continuation just means recording the top of the suspended stack and switching the stack pointer; resuming the continuation then restores the saved stack pointer. We thus focus on the first operation: handler lookup. With lexical effect handlers, a raise site is statically associated with a lexically scoped variable identifying a handler installed in the current stack. The job of handler lookup is to find this handler.

A running example. To illustrate the semantics of lexical handlers, consider the program in Figure 3.3. For simplicity, we assume that the effect type E has only one operation `op : unit -> unit`. Figure 3.4 shows the stack snapshots at four key program points.

When control enters a `handle` expression, a new handler instance is installed on the stack: the runtime allocates a new stack segment for the handler and assigns a globally unique name to the newly installed handler. These dynamically generated names are the run-time representation of the lexically scoped variables bound by `handle` expressions (`e0`, `e1`, `e2`, `e3`, and `e4` in Figure 3.3). The first snapshot in Figure 3.4 shows the stack at program point [A](#). Four handlers have been installed at this point, and they are assigned names 0, 1, 2, and 3. The runtime can choose these names arbitrarily as long as they are

```

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

```

Figure 3.3: A program with five handlers. The implementations of the handlers, except `e2`'s, are elided. The variables `e0`–`e4` are bound by the `handle` constructs and are available in the scope of the corresponding handled computations.



Figure 3.4: Stack snapshots at four program points during the execution of the program from Figure 3.3.

unique; we use small integers for illustration. The current stack consists of four segments. The arrow between two segments indicates the control flow when the computation on the deeper segment finishes and returns to the shallower one.

Next, upon `raise e2.op ()` on line 5, the runtime performs a handler lookup to find the handler that the variable `e2` refers to. Since `e2` has been substituted with the name 2, the runtime searches the stack for a handler with that name. Once found, the stack is unwound up to that handler, capturing a resumption stack consisting of the two unwound segments, and control jumps to program point **B**. The second diagram in Figure 3.4 captures the state of the stacks at this point.

As control enters the `handle` expression on line 9 (program point **C**), a new stack segment is allocated and a new handler instance (with name 4) is installed. The state of the stacks at this point is shown in the third diagram of Figure 3.4.

Next, on line 10, the captured continuation `k2` is resumed, so the suspended resumption stack is reinstated on top of the current stack. Control transfers to the program point immediately after the raise site (i.e., program point $\textcircled{D}$). The rightmost diagram in Figure 3.4 shows the stack at this point.

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

A more efficient implementation uses handlers' stack addresses as their identities [33, 37], as introduced in Chapter 2. Stack addresses are taken at the point of handler installation and passed down to raise sites. Handler lookup then becomes a constant-time operation: the runtime simply switches the stack pointer to the stack address passed down to the raise site. We call this implementation *address-based*.

The problem with identifying handlers by stack addresses. Unfortunately, this address-based implementation is simply incorrect in the presence of multishot resumptions. Suppose that the resumption `k2` needs to be resumed twice (e.g., `k2` is resumed again on line 11). The runtime must make a copy of the stack segments belonging to `k2`. In Figure 3.5, the snapshots in the top row are captured at program point $\textcircled{D}$ after a copy of the resumption stack is made and installed. In the search-based implementation, the handlers installed in the copied stack segments retain their original names (2 and 3). In contrast, in the address-based implementation, the handlers in the copied segments are associated with new addresses (`0x12` and `0x13`).

Immediately after program point $\textcircled{D}$, the program executes `raise e3.op ()` on line 6. In Figure 3.5, the snapshots in the bottom row are captured immediately after control transfers to a handler. In the search-based implementation, the runtime correctly finds the handler with name 3 via stack traversal and switches to the stack segment immediately enclosing it. In the address-based implementation, however, program point $\textcircled{D}$ still holds the stale address `0x03`, which is no longer valid—the stack segment associated with that address is not even part of the current stack. As a result, the runtime switches to the wrong segment. A possible remedy is to scan the copied stack segments and update corrupted references, but as mentioned earlier, this method is costly.

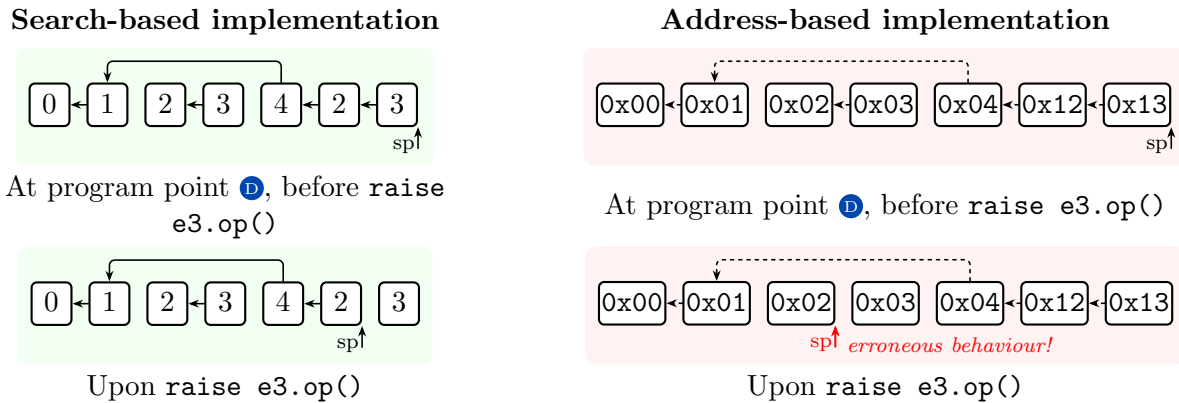


Figure 3.5: Stack snapshots before and after raising an effect within a copied resumption.

3.3 Virtualizing Stack Space

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

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

Zones and virtual addresses. Stack segments live in *zones*, managed by the sMMU. The sMMU ensures that copies of the same stack segment live in distinct zones: each time a copied resumption is resumed, a new zone is allocated to hold the copied stack segments. Figure 3.6 shows that a new zone (Zone 1) is allocated when the resumption `k2` is copied and resumed.

³Terminology note: we use the term *virtual* to refer exclusively to the stack-space virtualization provided by our runtime system. Beneath this layer of abstraction, the OS may also virtualize physical memory. Since OS virtual memory is conceptually transparent to our design, we do not consider it until Section 3.6.

Stack snapshot at program point [D](#)
in an implementation based on virtual resumptions

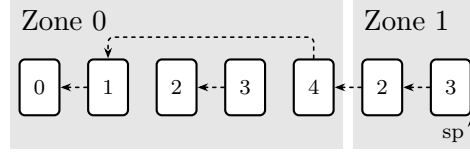


Figure 3.6: A stack snapshot with virtual resumptions. For clarity, stack segments are marked with handler IDs only; full virtual addresses can be reconstructed by prepending the zone number and appending the offset.

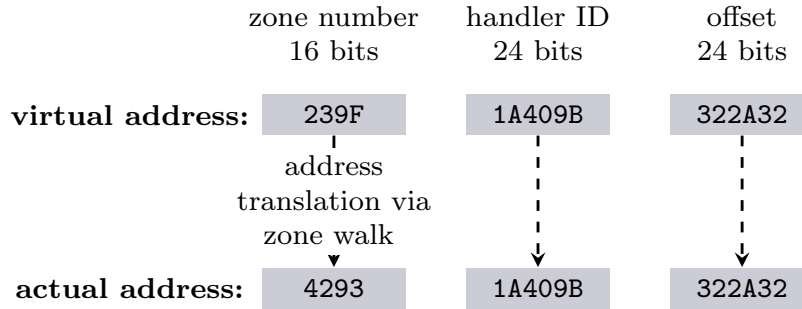


Figure 3.7: Structure of virtual addresses in the implementation described in Section 3.6. Address translation finds the actual zone number via zone walk.

One way to implement the sMMU is to allocate each zone as a large contiguous memory region and to allocate stack segments within it as fixed-size blocks. We implement this approach in our compiler and runtime system (Section 3.6.1), but it is possible to implement zones and segments using other allocation strategies (Section 3.6.6). The formalism in Section 3.4 gives a more abstract account that is independent of these implementation details.

In our canonical implementation, a virtual address has three fields: the higher bits are the *zone number*, the middle bits the *handler ID*, and the lower bits the *offset* (see Figure 3.7). Upon entering a `handle` expression, the sMMU allocates a new stack segment in the current zone using the next available handler ID. The current zone number, the new handler ID, and offset 0 together form a virtual address, which marks the installation point of the handler; it is passed down the stack as the capability for invoking operations implemented by that handler. The next available handler ID is drawn from an incrementing global counter initialized to zero at program start. The offset field is used to index into the stack storage local to the handler.

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

Address translation via zone walk. The use of virtual addresses necessitates address translation, which locates the zone containing the target handler through a process called *zone walk*. Upon a **raise**, the sMMU receives a virtual address to translate. First, it extracts the handler ID H from the virtual address. It then determines the current zone number using the stack pointer **sp**. After that, it walks the linked list of zones backwards, starting from the current zone.

Zone walk is performed at the granularity of zones. As the runtime traverses each zone, it does *not* examine every handler in the zone and compare its ID with H , which would be too expensive. Instead, it uses a simple *presence check*: for a given zone, if the first handler in the zone has ID H' such that $H \geq H'$, then the zone must contain handler H , so the walk stops; otherwise, the walk continues to the previous zone. It may be surprising that this simple check suffices. Its correctness, however, rests on two key properties of the language: (1) the runtime generates monotonically increasing handler IDs, and (2) handlers in the program text are lexically scoped. These properties allow us to establish the correctness of zone walk in Section 3.5.

Once the zone walk stops, say in a zone Z' , the sMMU completes the address translation: it modifies the original virtual address by replacing the zone number Z with Z' (see Figure 3.7). This actual address is then returned to the program for use in switching the stack pointer.

Demonstration. The stack snapshot in Figure 3.6 is taken at program point **D** right after the resumption **k2** is copied and resumed. Upon **raise e3.op ()** on line 6, the sMMU receives a virtual address—zone number $Z = 0$, handler ID $H = 3$ —to translate. From the current **sp**, the sMMU determines that the current zone number is 1. The sMMU then begins the zone walk. It first checks whether Zone 1 contains a handler with ID 3 using the presence check: since the first handler in Zone 1 has ID 2, and $H \geq 2$, the sMMU concludes that Zone 1 contains the desired handler. The sMMU immediately constructs the actual address by replacing the zone number Z in the virtual address with 1 and returns it.

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

Correctness and efficiency. An attentive reader may notice that the zone walk would seem to fail when translating a virtual address with zone number 0 and handler ID 4: because the first handler in Zone 1 has ID 2, and $4 \geq 2$, the walk would stop in Zone 1—but Zone 1 has no handler with ID 4! Rest assured, this scenario is not actually possible. While Figure 3.6 might give the impression that code running in Zone 1 could access the handler with ID 4, the type system prevents this possibility, thanks to the lexical scoping of handlers. Zone 1 is created for a copy of the resumption `k2`, which is captured between program points [A](#) and [B](#). At that point, the variable `e4` is not lexically in scope, so the handler with ID 4, which is only installed later, is inaccessible from within the resumption `k2`.

Zone walk is efficient. If the program does not use multishot resumptions, our implementation creates no new zones (Section 3.6.2), so all stack segments live in a single zone—zone walk reduces to a constant-time address-based lookup. When multishot resumptions are used, the overhead of zone walk is proportional to the number of zones traversed. Even so, zone walk remains more efficient than the search-based alternative, since traversal proceeds zone by zone rather than segment by segment, and the number of zones traversed is typically a small constant in practice.

Virtual resumptions, unlike existing implementations of lexical effect handlers [33, 37], allow multiple copies of the same resumption to be actively running at the same time, as address translation through zone walk does not depend on centralized resources. This ability has a happy consequence: the many copies of the same resumption can run in parallel without interfering with each other, which may translate to significant performance gains on multicore machines.

At this point, the reader may not yet believe that zone walk is guaranteed to find the correct handler, especially because the presence check may appear too coarse. In Section 3.5, we prove that virtual resumptions indeed implement the semantics of lexical effect handlers. But first, let us formalize the ideas in Section 3.4.

3.4 A Formal Model of the Zone-Based Semantics

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

The two semantics differ only in their treatment of continuation delimiters. ZL introduces a new form of delimiter—the zone delimiter—which marks zone boundaries. Section 3.5 proves a simulation theorem between SL and ZL, and Section 3.6 covers implementation details not modelled by ZL.

3.4.1 Syntax

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

value	$v ::= x \mid () \mid \lambda x. e$
handler value	$h ::= \lambda(y, k). e$
expression	$e ::= v \mid e_1 e_2 \mid \mathbf{handle} \lambda x. e \mathbf{with} h \mid \mathbf{raise} e_1(e_2) \mid \mathbf{resume} e_1(e_2)$

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

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

3.4.2 Operational Semantics

Auxiliary syntax. Figure 3.8 shows the auxiliary syntax used by the abstract machine. Syntax specific to ZL is shown in orange; the rest is shared by both semantics. The syntax

value	v	$::=$	$\dots \mid \mathbf{lbl}(n) \mid \mathbf{cont} K$
delimiter	ω	$::=$	$\bowtie \mid \blacktriangleright$
frame	A	$::=$	$\square e \mid v \square \mid \mathbf{raise} \square(e) \mid \mathbf{raise} v(\square) \mid \mathbf{resume} \square(e) \mid \mathbf{resume} v(\square)$
stack	K	$::=$	$\epsilon \mid K :: A \mid K :: \omega_h^n$
zone	Z	$::=$	$\epsilon \mid Z :: A \mid Z :: \bowtie_h^n$
configuration	M	$::=$	$\langle K \parallel e \parallel m \rangle \mid \langle K \parallel K' \parallel n \parallel v \parallel m \rangle$

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

Figure 3.8: Auxiliary syntax for the operational semantics. Elements in orange are specific to **ZL**; the rest are shared by both semantics.

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

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

A *zone* Z is a special case of a stack K . A stack may contain both standard delimiters $\bowtie$ and zone delimiters $\blacktriangleright$, while a zone may contain only standard delimiters $\bowtie$. A zone delimiter $\blacktriangleright$ marks the beginning of a new zone, so we abuse terminology and refer to both Z and $\blacktriangleright_h^n :: Z$ as zones. The auxiliary function $\mathcal{L}$ computes the set of labels present in a stack.

Reduction rules. Figure 3.9 presents two small-step abstract-machine semantics for the core language. The rules shared by both semantics are shown in black, those specific to **ZL**

APP	$\langle K \parallel e_1 e_2 \rangle$	$\longrightarrow \langle K :: \square e_2 \parallel e_1 \rangle$
BETA	$\langle K \parallel (\lambda x. e) v \rangle$	$\longrightarrow \langle K \parallel e\{v/x\} \rangle$
ENTER	$\langle K \parallel \text{handle } \lambda x. e \text{ with } h \parallel m \rangle$	$\longrightarrow \langle K :: \bowtie_h^m \parallel e\{\text{lbl}(m)/x\} \parallel m+1 \rangle$
LEAVE1	$\langle K :: \omega_h^n \parallel v \rangle$	$\longrightarrow \langle K \parallel v \rangle$
LEAVE2	$\langle K :: A \parallel v \rangle$	$\longrightarrow \langle K \parallel A[v] \rangle$
RAISE	$\langle K_1 :: \bowtie_h^n :: K_2 \parallel \text{raise lbl}(n)(v) \rangle$ where $n \notin \mathcal{L}(K_2)$	$\longrightarrow \langle K_1 \parallel \bowtie_h^n :: K_2 \parallel n \parallel v \rangle$
RAISE	$\langle K \parallel \text{raise lbl}(n)(v) \rangle$	$\longrightarrow \langle K \parallel \epsilon \parallel n \parallel v \rangle$
UNW-INTER-ZONE	$\langle K :: \blacktriangleright_h^l :: Z \parallel K' \parallel n \parallel v \rangle$ where $l \geq n$	$\longrightarrow \langle K \parallel \blacktriangleright_h^l :: Z :: K' \parallel n \parallel v \rangle$
UNW-INTRA-ZONE	$\langle K :: \bowtie_h^n :: Z \parallel K' \parallel n \parallel v \rangle$	$\longrightarrow \langle K \parallel \blacktriangleright_h^n :: Z :: K' \parallel n \parallel v \rangle$
INVOKE	$\langle K \parallel \omega_h^n :: K' \parallel n \parallel v \rangle$ where $h = \lambda(y, k). e$	$\longrightarrow \langle K \parallel e\{v/y\}\{\text{cont } \omega_h^n :: K'/k\} \rangle$
RESUME	$\langle K \parallel \text{resume}(\text{cont } K')(v) \rangle$	$\longrightarrow \langle K :: K' \parallel v \rangle$

Figure 3.9: Reduction rules of SL and ZL. Rules in blue are specific to SL, those in orange are specific to ZL, and the rest are shared. The counter m is omitted when not relevant.

are shown in orange, and those specific to SL are shown in blue. All administrative rules are omitted for brevity, except for APP, which is shown as an example.

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

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

ZL's RAISE rule does not immediately yield an invoke-ready configuration. Instead, to model zone walk (Section 3.3), it simply transitions the machine state into unwind mode, initiating an unwinding process that traverses the list of zones to find the desired handler. UNW-INTER-ZONE advances the traversal one zone at a time. If the current zone is guarded

by a zone delimiter whose label l is greater than or equal to the desired label n , the zone $\blacktriangleright_h^l :: Z$ is popped off the active stack and pushed onto the resumption stack. In particular, if $l = n$, the resulting configuration is invoke-ready, so zone walk ends here; otherwise, zone walk continues. **UNW-INTRA-ZONE** is in effect when the current zone contains a handler with label n . Crucially, the delimiter $\bowtie$ of the handler is upgraded to $\blacktriangleright$, marking the creation of a new zone for the captured resumption. The resulting configuration is now invoke-ready, so zone walk concludes here as well.

Discussion. Although **SL** finds the desired handler in just one reduction step, a proper implementation of this semantics must still examine, one by one, the labels attached to each delimiter ($\bowtie$ or $\blacktriangleright$) on the stack. In contrast, although handler lookup in **ZL** may take multiple unwinding steps, the number of steps is proportional only to the number of zone delimiters ($\blacktriangleright$), which are typically far sparser on the stack than standard delimiters ($\bowtie$). Also notice that **ZL** does not model zone numbers: their exact values are immaterial to the high-level semantics of virtual resumptions. Instead, **ZL** models zone boundaries using zone delimiters, and address translation is modelled as searching the stack for the right zone boundary via zone walk.

Demonstration. Consider the stack diagram in Figure 3.6. The shape of the same *active* stack in **ZL** is $\epsilon :: \bowtie^0 :: \bowtie^1 :: \bowtie^4 :: \blacktriangleright^2 :: \bowtie^3$ (non-handler frames and handler implementations are omitted for brevity). In a first example, the effect is raised to a handler with ID 3. The reduction steps taken in **ZL** are **RAISE**, **UNW-INTRA-ZONE**, and then **INVOKE**—the desired handler is immediately found in the current zone. In a second example, the effect is raised to a handler with ID 1. The reduction steps taken in **ZL** are **RAISE**, **UNW-INTER-ZONE**, **UNW-INTRA-ZONE**, and **INVOKE**—first walking out of the current zone and then finding the desired handler in the enclosing zone.

3.5 Correctness

We now prove that **ZL** preserves the semantics of **SL**. This result formally establishes the guarantee that a program written with **SL** in mind behaves the same when run under **ZL**. It is stated in Theorem 3.1, which says that if a program e type-checks and evaluates to a value v under **SL**, then it also evaluates to the same value v under **ZL**. Recall that **SL** and **ZL** share the same source syntax.

Theorem 3.1 (Semantic Preservation). *If $\epsilon \mid \epsilon \vdash e : \tau \mid \epsilon$ and $\langle \epsilon \parallel e \parallel 0 \rangle \longrightarrow^* \langle \epsilon \parallel v \parallel m \rangle$, then $\langle \epsilon \parallel e \parallel 0 \rangle \longrightarrow^* \langle \epsilon \parallel v \parallel m \rangle$.*

Theorem 3.1 follows easily from a simulation lemma (Lemma 3.2), which states that every reduction step in SL can be simulated by a sequence of steps in ZL.

Lemma 3.2 (Simulation). *Given an SL configuration M and a ZL configuration M such that $\vdash M$ and $M \sim M$, if $M \longrightarrow M'$, then there exists a ZL configuration M' such that $M \longrightarrow^* M'$ and $M' \sim M'$.*

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

3.5.1 Relating SL and ZL

Lemma 3.2 is stated in terms of a relation $M \sim M$ between SL and ZL configurations. This relation is defined inductively over the syntax, which further requires relations over values, expressions, handler values, frames, and stacks: $v \stackrel{\text{val}}{\sim} v$, $e \stackrel{\text{exp}}{\sim} e$, $h \stackrel{\text{hdl}}{\sim} h$, $A \stackrel{\text{frm}}{\sim} A$, and $K \stackrel{\text{ctx}}{\sim} K$. Since the syntaxes of SL and ZL largely coincide, most of the rules are trivial and are omitted. The only nontrivial case is the relation on stacks, the two key rules of which are

$$\frac{K \stackrel{\text{ctx}}{\sim} K \quad h \stackrel{\text{hdl}}{\sim} h}{K :: \bowtie_h^n \stackrel{\text{ctx}}{\sim} K :: \bowtie_h^n} \qquad \frac{K \stackrel{\text{ctx}}{\sim} K \quad h \stackrel{\text{hdl}}{\sim} h}{K :: \bowtie_h^n \stackrel{\text{ctx}}{\sim} K :: \blacktriangleright_h^n}$$

These rules state that both forms of ZL delimiter, $\bowtie_h^n$ and $\blacktriangleright_h^n$, are related to the SL delimiter $\bowtie_h^n$.

3.5.2 Type System

Theorem 3.1 is stated for well-typed programs. Our type system is similar to prior type systems for lexical handlers [68, 70, 4, 6]. It applies to both SL and ZL and covers both source programs and run-time configurations. The syntax of type-level constructs and a selection of typing rules are shown in Figures 3.10 and 3.11.

In addition to types, *labels* and *capabilities* are also at the type level; we call them *constructors* collectively. A constructor context Δ tracks the kinds of *constructor variables*. A label η is either a label variable ρ or a label constant n ; label constants exist only at run time. A capability set C is composed of capability variables α and mappings from labels η to effect names F . For a program point that provides capabilities C , the presence of a mapping $\eta \mapsto F$ in C indicates that a handler for the F effect with label η is accessible from that program point. A function type $\forall[\Delta \mid C]. \tau_1 \rightarrow \tau_2$ abstracts over constructor variables (bound in Δ) and mentions a precondition C describing the capabilities required to be present on the stack when the function is called. Dually, C can be seen as the side effects the function may perform. Similarly, for a continuation of type $\text{cont}[C]. \tau_1 \rightarrow \tau_2$, capabilities C must be present when it is invoked.

The typing rules are implicitly parameterized over a global map $\mathbb{F}$ from effect names to their signatures. Expression typing judgements have the form $\Delta \mid \Gamma \vdash e : \tau \mid C$, where C is the capabilities required to be present when evaluating e ; the capabilities C can also be seen as the side effects that e may perform. Value typing judgements have the form $\Delta \mid \Gamma \vdash v : \tau$; values are free of side effects. Lambda abstractions are implicitly polymorphic in constructor variables (bound by Δ' in the typing rule). Label values $\text{lbl}(n)$ are given singleton types $\text{LBL}(n)$, and continuations are typed based on the type of the captured stack K .

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

In the typing rule for **handle** $\lambda x. e$ **with** h , the expression e is parameterized over a label variable ρ and a term variable x of type $\text{LBL}(\rho)$. Variable x stands for the label value that will be dynamically created when the handler is installed. Variable ρ appears in the mapping $\rho \mapsto F$ in the capability context of e , which states the requirement that a handler for F be available when evaluating e . The premise $\Delta \mid \Gamma \vdash h : F \# \tau \mid C$ says that the handler value h implements the F effect and, when invoked, performs a computation having type τ and requiring capabilities C . The premise $\Delta \vdash \tau$ says that the type τ of the expression is well-formed without ρ in the constructor context, effectively preventing ρ and x from escaping. This condition is standard; type systems for lexical handlers [68, 4] or lexical regions [57, 10, 19] use it to ensure type safety. In the typing rule for **raise** $e_1(e_2)$, e_1 is given the singleton type $\text{LBL}(\eta)$, and the capability context C must provide the capability to use η . The use of singleton types for dynamically created resources, as well as the separation between labels and the capabilities to use them, is similar to alias types [61]. In the typing rule for **resume** $e_1(e_2)$, the capabilities required to invoke the continuation e_1 must be present in the current capability context C .

constructor kind	$\kappa ::= \mathbf{Label} \mid \mathbf{Cap}$
constructor	$c ::= \eta \mid C$
label	$\eta ::= \rho \mid n$
capability set	$C ::= \epsilon \mid C \cup \alpha \mid C \cup \{\eta \mapsto F\}$
constructor var	$\beta ::= \rho \mid \alpha$
constructor context	$\Delta ::= \epsilon \mid \Delta, \beta : \kappa$
term-var context	$\Gamma ::= \epsilon \mid \Gamma, x : \tau$
type	$\tau ::= \mathbf{unit} \mid \mathbf{LBL}(\eta) \mid \forall[\Delta \mid C]. \tau \rightarrow \tau \mid \mathbf{cont}[C]. \tau \rightarrow \tau$

$$\text{dom}(\epsilon) = \emptyset \quad \text{dom}(C \cup \{n \mapsto F\}) = \text{dom}(C) \cup \{n\}$$

$$\boxed{\Delta \mid \Gamma \vdash v : \tau} \quad (\text{value typing})$$

$$\frac{\Delta, \Delta' \mid \Gamma, x : \tau_1 \vdash e : \tau_2 \mid C}{\Delta \mid \Gamma \vdash \lambda x. e : \forall[\Delta' \mid C]. \tau_1 \rightarrow \tau_2} \quad \frac{}{\Delta \mid \Gamma \vdash \mathbf{lbl}(n) : \mathbf{LBL}(n)}$$

$$\frac{\vdash K : [C']\tau_1 \rightsquigarrow [C]\tau_2}{\Delta \mid \Gamma \vdash \mathbf{cont} K : \mathbf{cont}[C]. \tau_1 \rightarrow \tau_2}$$

$$\boxed{\Delta \mid \Gamma \vdash e : \tau \mid C} \quad (\text{selected rules})$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : (\forall[\overline{\beta} : \kappa \mid C]. \tau_1 \rightarrow \tau_2) \mid C \quad \Delta \mid \Gamma \vdash e_2 : \tau_1 \{\bar{c}/\bar{\beta}\} \mid C \quad \Delta \vdash \bar{c} : \bar{\kappa}}{\Delta \mid \Gamma \vdash e_1 e_2 : \tau_2 \{\bar{c}/\bar{\beta}\} \mid C} \quad \frac{\Delta, \rho : \mathbf{Label} \mid \Gamma, x : \mathbf{LBL}(\rho) \vdash e : \tau \mid C \cup \{\rho \mapsto F\} \quad \Delta \mid \Gamma \vdash h : F \# \tau \mid C \quad \Delta \vdash \tau}{\Delta \mid \Gamma \vdash \mathbf{handle} \lambda x. e \mathbf{with} h : \tau \mid C}$$

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

Figure 3.10: Type-level constructs and selected typing rules for values and expressions. The type system works for both **SL** and **ZL**. The full set of rules can be found in Appendix B.

$$\boxed{\Delta \mid \Gamma \vdash h : F \# \tau \mid C} \quad (\text{handler typing})$$

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

$$\boxed{\vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C} \quad \boxed{\vdash K : [C_1]\tau_1 \rightsquigarrow [C_2]\tau_2} \quad (\text{frame and stack typing})$$

$$\frac{\vdash K : [C_2]\tau_2 \rightsquigarrow [C_3]\tau_3 \quad \vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C_2}{\vdash K :: A : [C_2]\tau_1 \rightsquigarrow [C_3]\tau_3} \quad \frac{\vdash K : [C_1]\tau_1 \rightsquigarrow [C_2]\tau_2 \quad \epsilon \mid \epsilon \vdash h : F \# \tau_1 \mid C_1 \quad \forall l \in \text{dom}(C_1), l < n}{\vdash K :: \omega_h^n : [C_1 \cup \{n \mapsto F\}]\tau_1 \rightsquigarrow [C_2]\tau_2}$$

Figure 3.11: Selected typing rules for handlers, frames, and stacks.

Stack typing judgements have the form $\vdash K : [C_1]\tau_1 \rightsquigarrow [C_2]\tau_2$. The type indicates that K can be installed on a stack that expects in its hole a computation of type τ_2 with capability context C_2 , and that K itself expects in its hole a computation of type τ_1 with capability context C_1 . To type a stack of the form $K :: A$, the rule ensures that the output type and capability context of the frame A match the input type and capability context of the stack K . Notice that the frame type of A does not have an input capability context, as non-handler frames do not introduce capabilities.

The most consequential rule for our purposes is the stack-typing rule for $K :: \omega_h^n$. It additionally requires that any capability needed by the computation to be installed in the hole of $K :: \omega_h^n$ be arithmetically no larger than n —that is, created no later than label n . This invariant holds because, statically, the computation can only access labels lexically in scope, and, dynamically, labels in the computation’s lexical context are assigned monotonically increasing numbers when created. This invariant is exactly what makes the coarse presence check of zone walk correct. Configuration typing $\vdash M$ is standard and thus omitted.

With the type system in place, the type-safety theorem for SL is standard [68, 4, 6].

Theorem 3.3 (SL Type Safety). (Progress) *If $\vdash M$, then either M is a final configuration or there exists M' such that $M \longrightarrow M'$.* (Preservation) *If $\vdash M$ and $M \longrightarrow M'$, then $\vdash M'$.*

We omit the proof.

3.5.3 Proving Simulation

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

$$\mathbf{Acc}_m(K) = \{ l < m \mid \text{label } l \text{ is installed on } K \text{ and accessible from the top of } K \}$$

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

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

As a demonstration, consider again the active stack in Figure 3.6, whose shape in ZL is $\epsilon :: \blacktriangleright^0 :: \blacktriangleright^1 :: \blacktriangleright^4 :: \blacktriangleright^2 :: \blacktriangleright^3$. The set of labels accessible from the top of this stack is $\{0, 1, 2, 3\}$:

$$\begin{aligned} \mathbf{Acc}_\infty(\epsilon :: \blacktriangleright^0 :: \blacktriangleright^1 :: \blacktriangleright^4 :: \blacktriangleright^2 :: \blacktriangleright^3) &= \{ l \in \{2, 3\} \mid l < \infty \} \cup \mathbf{Acc}_{\min(\infty, 2)}(\epsilon :: \blacktriangleright^0 :: \blacktriangleright^1 :: \blacktriangleright^4) \\ &= \{ l \in \{2, 3\} \mid l < \infty \} \cup \{ l \in \{0, 1, 4\} \mid l < 2 \} = \{0, 1, 2, 3\}. \end{aligned}$$

Notice that label 4, installed by $\blacktriangleright^4$, is *not* accessible from the stack top, because of the zone delimiter $\blacktriangleright^2$ in between. This is precisely why the apparent failure of zone walk discussed in Section 3.3 cannot occur. Indeed, if somehow $\blacktriangleright^4$ could be invoked during evaluation, then no unwinding rule in Figure 3.9 would apply, and thus evaluation would get stuck.

We state some useful lemmas about accessible labels. Lemma 3.4 can be proved by induction on K .

Lemma 3.4. *If $l \in \mathbf{Acc}_m(K)$, then $l < m$.*

Lemma 3.5 states that, for a well-typed stack K , all the labels K makes available to its hole are accessible from within the hole. It can be proved by induction on the typing derivation of K .

Lemma 3.5. *If $\vdash K : [C_1]\tau_1 \rightsquigarrow [C_2]\tau_2$, then $\text{dom}(C_1) \subseteq \mathbf{Acc}_\infty(K)$.*

Lemma 3.6 states that, in a well-typed configuration that is about to invoke a handler with label n , the label is accessible from the top of the current stack.

Lemma 3.6 (Reachability Invariant). *If $\vdash \langle K \parallel \text{raise lbl}(n)(v) \rangle$, then $n \in \mathbf{Acc}_\infty(K)$.*

Proof. Because the configuration is well-typed, we have

$$\vdash K : [C_1]\tau_1 \rightsquigarrow [C_2]\tau_2 \quad \text{and} \quad \epsilon \mid \epsilon \vdash \text{raise lbl}(n)(v) : \tau_1 \mid C_1.$$

By the well-typedness of K and Lemma 3.5, we have $\text{dom}(C_1) \subseteq \text{Acc}_\infty(K)$. By the well-typedness of the **raise** expression, $n \in \text{dom}(C_1)$. The result follows. $\square$

Lemma 3.7 states that if an SL configuration type-checks and a ZL configuration is related to it, then the ZL configuration type-checks as well. It can be proved by induction on the typing derivation of the SL configuration.

Lemma 3.7. *If $\vdash M$ and $M \sim M$, then $\vdash M$.*

We are now ready to prove the **RAISE** case of Lemma 3.2.

Proof of Lemma 3.2, case RAISE. The proof concerns the following simulation diagram:

$$\begin{array}{ccc} \langle K_1 :: \bowtie_h^n :: K_2 \parallel \text{raise lbl}(n)(v) \rangle & \longrightarrow_{\text{SL}} & \langle K_1 \parallel \bowtie_h^n :: K_2 \parallel n \parallel v \rangle \\ \sim & & \sim \\ \langle K_1 :: \omega_h^n :: K_2 \parallel \text{raise lbl}(n)(v) \rangle & \longrightarrow_{\text{ZL}}^* & \langle K_1 \parallel \bowtie_h^n :: K_2 \parallel n \parallel v \rangle \end{array}$$

The top row is the SL step by rule **RAISE**; the bottom row is the desired ZL reduction simulating it. The goal is to show that given the SL configurations on the top-left and top-right, and the ZL configuration on the bottom-left, if they satisfy the relations $\longrightarrow_{\text{SL}}$ (top row) and $\sim$ (left column), then there exists a ZL configuration on the bottom-right that satisfies $\longrightarrow_{\text{ZL}}^*$ (bottom row) and $\sim$ (right column). We show that such a bottom-right ZL configuration exists; its form is given in the diagram. It obviously satisfies the $\sim$ relation (right column). We now derive $\longrightarrow_{\text{ZL}}^*$. By rule **RAISE**, the bottom-left configuration first transitions into unwind mode: $M_1 = \langle K_1 :: \omega_h^n :: K_2 \parallel \epsilon \parallel n \parallel v \rangle$. There are two cases.

- K_2 contains no zone delimiters. Then M_1 takes a single step to the bottom-right configuration by rule **UNW-INTER-ZONE** or **UNW-INTRA-ZONE**, depending on whether ω_h^n is a zone delimiter.
- K_2 contains at least one zone delimiter: $K_2 = Z_0 :: \bowtie_{h_1}^{m_1} :: Z_1 :: \dots :: \bowtie_{h_k}^{m_k} :: Z_k$ ($k \geq 1$). By assumption the top-left SL configuration is well-typed, so by Lemma 3.7 the bottom-left ZL configuration is well-typed too. It follows from Lemma 3.6 that $n \in \text{Acc}_\infty(K_1 :: \omega_h^n :: K_2)$. Meanwhile, by the SL step and rule **RAISE**, $n \notin \mathcal{L}(K_2)$, hence $n \notin \mathcal{L}(K_2)$. Combining these, it must be that $n \in \text{Acc}_{\min(m_1, \dots, m_k)}(K_1 :: \omega_h^n :: Z_0)$, so by Lemma 3.4, $n < \min(m_1, \dots, m_k)$. Since all zone delimiters in K_2 have labels greater than n , rule **UNW-INTER-ZONE** applies repeatedly, unwinding the zones from $\bowtie_{h_k}^{m_k} :: Z_k$ down to $\bowtie_{h_1}^{m_1} :: Z_1$, until the walk reaches Z_0 ; then a single step takes the configuration to the bottom-right configuration, as in the base case. $\square$

3.6 Implementation

We integrate the idea of virtual resumptions into the LEXA compiler [55]. LEXA is a simple functional language featuring lexical effect handlers. Its compiler translates a LEXA program into C, using low-level stack-switching routines (packaged as a C library, called **StackTrek**) to implement effect handlers. The compiled C program is fed to LLVM for optimization and code generation, and a tracing garbage collector manages both heap and stack memory. **StackTrek** consists of x86 assembly functions for stack switching and macros **HANDLE**, **RAISE**, and **RESUME** that use those functions to install a handler, raise an effect, and resume a resumption. The compiler uses stack addresses to represent handler labels, enabling constant-time handler lookup. However, Base LEXA cannot support multi-shot resumptions properly, as copying stack segments to new memory locations invalidates references to handlers installed on the copied stack.

3.6.1 Overview

To integrate full support for multishot resumptions into LEXA, we implement in C the sMMU described in Section 3.3. The sMMU manages the stack space using Linux’s system calls for virtual memory management: zones are created by reserving 512 KB chunks of aligned virtual address space, and stack segments are allocated as fixed-size blocks of 8 KB by committing the relevant memory range to physical memory. Thus, a zone can hold up to 64 stack segments; this number can be freely adjusted by changing the zone sizes. This allocation strategy lets the sMMU maintain an ordered memory layout without unnecessary physical cost.

Figure 3.12 illustrates the memory layout of zones and the stack segments within them. The LEXA compiler allocates a *header frame* at the base of each stack segment. A header frame contains information about the handler and points to the previous stack segment in the active stack (dashed arrows in the diagram). We use a flag in the header frame to indicate whether the stack segment is the start of a zone; we call such a flagged stack segment a *zone header*. A zone header points to the previous zone header in the active stack. In the diagram, zone headers are marked with a black circle in the top-left corner, and the solid arrow shows the link between them.

The design in Section 3.3 calls for a global counter to track the next available handler ID. We observe that a monotonically increasing counter is not necessary; it suffices that a freshly allocated handler ID is greater than any handler ID currently in use. So, as an optimization, the sMMU recycles handler IDs using a global bitmap that tracks the handler

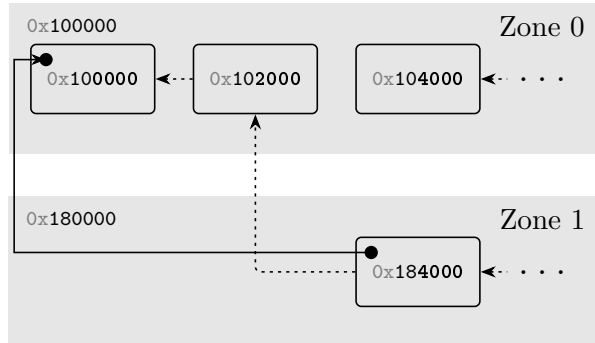


Figure 3.12: Layout of stack segments using OS virtual memory. Grey backgrounds represent zones. Black circles mark zone headers; the solid arrow links zone headers, while dashed arrows link consecutive stack segments.

IDs in use. The sMMU maintains another global bitmap to cache zones that have been allocated and then freed; as a result, the sMMU operations usually avoid system calls.

Stack switching using the sMMU API. The sMMU manages stack allocation and address translation. It exposes the API functions `get_stack`, `free_stack`, `translate_address`, and `dup_resumption`. The `StackTrek` library in the LEXA runtime is surgically modified (see Figure 3.13) to fully support multishot resumptions using this API.

`get_stack` takes the stack pointer `sp`, identifies the current stack segment’s zone number and handler ID through `sp`, and allocates a new stack segment in the current zone with the next available handler ID. It is used on line 2, in the `HANDLE` macro, to allocate a new stack segment when installing a handler. `free_stack` releases the memory associated with a stack segment, recycling the handler ID (and possibly the zone); it is called on line 5 upon control leaving the handler’s lexical scope. `dup_resumption` walks through the stack segments in the resumption and copies them into fresh zones (line 14). `translate_address` performs the address translation described in Section 3.3 and is used by the `RAISE` macro on line 8.

3.6.2 Single-Shot and Last-Shot Optimizations

Optimizations exist in Base LEXA [33] that avoid stack copying and stack allocation; our new implementation preserves them. A single-shot resumption is guaranteed not to be resumed again, so it is safe to destructively resume it in place. A multishot resumption can

```

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

```

Figure 3.13: Pseudocode implementation of the `StackTrek` macros using the sMMU API. The calls into the sMMU mark the main changes from Base LEXA. A detailed account of stack switching with `StackTrek` appears in Chapter 2 and in [33].



Figure 3.14: Intra-zone header elision. Pink stack segments mark potential locations of an in-place resumption. If the resumption starts at a greater handler ID than that of the closest zone header (black circle), the intra-zone header can be elided.

also be resumed in place if it is the last time it will be resumed; LEXA has a `resume_final` construct that lets the programmer indicate that a resumption is the last shot. In principle, reference counting could determine when a resumption is used for the last time; we leave this for future work, as it is orthogonal to our focus.

Resuming in place introduces a complication: it potentially installs a zone header in the middle of a zone. To support this case while minimizing the overhead of zone walks, we introduce an optimization called *intra-zone header elision*. First, the sMMU uses a global pointer to track the closest zone header instead of relying on `sp`. Then, if the in-place resumption lives in the same zone as this zone header and starts at a handler ID greater than the header’s, the runtime does not install the intra-zone header. Figure 3.14 shows an example memory layout: the pink stack segments represent the potential locations of the resumption. If the resumption starts at a greater handler ID than the closest zone header’s, the sMMU can skip installing the intra-zone header; if it starts at a smaller handler ID, the intra-zone header must be installed to establish a new minimum ID for presence checks during future zone walks.

To fully exploit this optimization, the sMMU treats the main stack as a special zone header at zone 0 with a handler ID smaller than any valid handler ID. This means that if a program uses only single-shot resumptions, no extra zone header is installed, and handler lookup is always constant-time. The empirical results in Section 3.7 confirm that our approach incurs minimal overhead, suggesting that the optimizations for single- and last-shot resumptions are preserved.

3.6.3 Optimizations for Abortive and Tail-Resumptive Handlers

A *tail-resumptive* handler invokes the captured resumption in tail position, and an *abortive* handler does not resume the resumption at all. LEXA optimizes both: a tail-resumptive handler is invoked in place like a regular function call, whereas an abortive handler simply aborts the surrounding computation delimited by the header frame. Because there is no

need to capture and restore a resumption, the compiler can avoid allocating a new stack segment and simply allocate the header frame in the middle of the current stack. Our new runtime supports these optimizations without extra effort: even with header frames located in the middle of a stack segment, the sMMU performs the same address translation as for regular handlers to locate the correct handler instances.

3.6.4 Stack-Allocated Mutable State

Stack-allocated mutable state is challenging to implement in the presence of multishot resumptions: as each copy of the stack maintains its own version of the mutable state, care must be taken to ensure that mutations are performed on the correct version. Our approach can be adapted to support stack-allocated mutable state by employing the same zone-walking strategy to find the correct copy of the state. This adaptation requires distinguishing stack addresses from heap addresses (either at compile time or efficiently at run time), so that only stack addresses are translated. While we leave full support to future work, our implementation supports a restricted form known as *handler-local* mutable state—state belonging to a handler instance and accessible through the handler reference. Since handler-local state is allocated in the handler’s header frame, locating the correct version of the state is equivalent to locating the correct copy of the handler instance by walking zones.

3.6.5 Support for Simple Parallelism

As discussed in Section 3.3, our implementation strategy enables parallel execution of multiple copies of the same resumption. To support this, we extend LEXA with a parallel-let expression in the style of Multilisp [21] that evaluates two subexpressions in parallel and binds their results to two variables. Its syntax and operational semantics are given in Appendix B.5. Our implementation realizes parallel-let using pthreads and statically enforces that no effect escapes either of the parallel branches.

We implement a simple fork-join thread pool with a global work queue. A thread that encounters a parallel-let expression forks the two branches into the work queue and joins them before proceeding. To prevent deadlock, a thread waiting for a join tries to help with other tasks in the work queue instead of idly waiting. While a more sophisticated scheduler with thread-local queues should yield better speedup, we leave that to future work. To prevent interference between threads, each thread is assigned a unique set of zones and maintains a thread-local pointer to the closest zone header; a thread-local bitmap tracks the

zones owned by the thread. A run-time error is raised if a thread tries to modify memory outside its assigned zones, such as by destructively resuming into another thread’s zone.

3.6.6 Other Ways to Implement Virtual Resumptions

The implementation described in Section 3.6.1 represents one way to realize virtual resumptions; other approaches are possible, each with its own trade-offs. In our approach, using virtual memory forces each stack segment to stay at a fixed address, which prevents relocation (e.g., allocating a larger stack when the original overflows); zones also have fixed sizes, which allows locating zones using pointer arithmetic. While modern computers should have enough virtual memory to accommodate our approach, there could be cases where virtual memory is not a viable option.

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

3.7 Evaluation

We assess the performance impact of integrating virtual resumptions into the LEXA compiler. We refer to the original LEXA compiler as *Base LEXA* and the new compiler as *Zone LEXA*.

Base LEXA has only limited support for multishot resumptions: resumptions cannot span more than one stack segment, handler-local mutable state is disallowed, and multiple copies of the same resumption cannot be active simultaneously. These restrictions simplify Base LEXA: instead of performing address translation to access the correct header frame, it saves a multishot resumption’s `sp` in a heap-allocated object independent of the header frames. Zone LEXA removes these restrictions and fully supports multishot resumptions without extra heap allocations, but inevitably introduces some overhead due to the additional bookkeeping needed to maintain an organized memory layout. This motivates our first research question.

RQ1. For benchmarks that Base LEXA can handle, does the additional bookkeeping in Zone LEXA introduce only small overhead compared to Base LEXA?

Base LEXA and the most recent version of the Effekt compiler [37] both implement lexical effect handlers through direct manipulation of the call stack, but both are limited in their support for multishot resumptions: different copies of the same resumption cannot be active simultaneously. This restriction is fundamentally incompatible with running multishot resumptions in parallel. Zone LEXA removes the restriction, which motivates our second research question.

RQ2. For benchmarks where multishot resumptions are embarrassingly parallel, does Zone LEXA unlock significant speedups by executing them in parallel?

3.7.1 Overhead of Zone LEXA vs. Base LEXA

We use a community-maintained benchmark suite designed to evaluate effect handler implementations [15], which primarily contains effect-heavy programs that stress the efficiency of handler invocation and resumption capture. We gather additional benchmarks, drawn from the literature on effect handlers, that exercise multishot resumptions. All but two are designed to run without handler-local state or extra handlers inside resumptions, so that Base LEXA can handle them. We include two other languages supporting lexical effect handlers—Effekt and Koka. Koka’s *named handlers* [65] are a form of lexical handlers; we use them (rather than Koka’s dynamically scoped handlers) when implementing the benchmarks in Koka. Performance variations among these systems stem from many factors, not just effect-handler efficiency; we include Effekt and Koka only as reference points. The main comparison is between Base LEXA and Zone LEXA.

Zero- and single-shot benchmarks. The first group of benchmarks in Table 3.1 is the community-maintained suite [15], excluding those that exercise multishot resumptions. Zone LEXA achieves results comparable to Base LEXA, adding little to no overhead on most benchmarks. For single-shot resumptions, Zone LEXA performs the single-shot optimization of Section 3.6.2, which obviates the need for zone walk and exposes further optimization opportunities to LLVM. The table also shows that LEXA is competitive with Effekt and Koka on these benchmarks.

Multishot benchmarks. The second group exercises multishot resumptions and comprises benchmarks from the community suite together with ones we gathered. In most benchmarks, Zone LEXA performs similarly to Base LEXA: this parity results from a trade-off between the cost of memory bookkeeping and the overhead saved by not heap-allocating

Table 3.1: Benchmark running times (milliseconds, mean) for four implementations of lexical effect handlers. The first group of benchmarks does not involve multishot resumptions; the second group does. No parallelism among multishot resumptions is exploited by Zone LEXA in this table; RQ2 is addressed in Section 3.7.2. “n/a” marks benchmarks an implementation cannot run. Averages are taken over the benchmarks that Base LEXA supports.

Benchmark	Base LEXA	Zone LEXA	Overhead	Effekt	Koka
	(ms)	(ms)	(%)	(ms)	(ms)
Countdown	1	0	−100.0	0	3529
Fibonacci Recursive	581	581	0.0	1853	1120
Product Early	100	100	0.0	255	1766
Iterator	10	9	−10.0	9	484
Generator	832	873	4.9	3434	7767
Resume Nontail	114	131	14.9	111	1778
Parsing Dollars	271	270	−0.4	2347	2886
Handler Sieve	480	478	−0.4	360	2181
Nqueens	280	349	24.6	403	1776
Triples	201	214	6.5	162	1717
Tree Explore	233	222	−4.7	260	249
Tree Explore BFS	76	78	2.6	182	123
Tree Generator	309	295	−4.5	572	2117
Expectimax	n/a	331	n/a	415	778
Knapsack	449	508	13.1	408	2674
Nim	304	299	−1.6	397	711
Checkpointing	n/a	120	n/a	124	n/a
Average	282.7	293.8	3.9	716.9	2058.5

resumption objects. Because most benchmarks invoke each captured resumption only two or three times, the two costs balance out. **Nqueens** performs backtracking search to find all solutions to the N-Queens problem; Zone LEXA incurs higher overhead here because, unlike other benchmarks, it invokes each resumption a number of times proportional to the input size, so the bookkeeping cost overwhelms the saved heap-allocation cost. **Expectimax** implements the expectimax search of Section 3.1; it installs two separate handlers per recursive call, and Base LEXA fails to support it because some resumptions span two stack segments. **Knapsack** is another backtracking search, adapted from [63]; the overhead of Zone LEXA here stems from a lack of inlining due to larger function bodies, and forcing the relevant functions to be inlined in the generated C recovers performance similar to Base LEXA. **Checkpointing**, from [37], exercises both multishot resumptions and stack-allocated mutable state; Base LEXA does not support such state, whereas Zone LEXA supports it using the handler-local state of Section 3.6.4. While Koka supports handler-local state and can in principle run this benchmark, it cannot handle any reasonable input size due to stack overflow.

Overall, the results show that Zone LEXA incurs at most moderate overhead on these multishot benchmarks relative to Base LEXA, while fully supporting multishot resumptions. Where the overhead is more noticeable (e.g., **Nqueens** and **Knapsack**), it is more than offset by parallel speedup, which we discuss next. Zone LEXA also exhibits the same asymptotic scaling behaviour as Base LEXA, indicating that zone walk has only constant overhead in practice, and—because it does not heap-allocate resumption objects—uses significantly less heap memory on some multishot benchmarks.

3.7.2 Speedup from Parallelizing Multishot Resumptions

Many applications of multishot resumptions are embarrassingly parallel: copies of the same resumption can be executed in parallel without synchronization or communication. Figure 3.15 shows the *strong scaling* of Zone LEXA on a selection of such benchmarks—fixing the workload and varying the number of threads. **Nqueens**, **Expectimax**, **Knapsack**, and **Nim** are straightforwardly parallelized versions of the benchmarks in Table 3.1, while **Tree Explore Det** is modified from **Tree Explore** to eliminate the need for synchronization. A cutoff depth is chosen manually for each benchmark to maximize speedup while limiting thread-scheduling overhead. All speedups are relative to a sequential baseline.

The plot shows that parallelization yields speedups for all the benchmarks. In contrast, neither Base LEXA nor Effekt admits such parallelism, due to the restrictions they impose on multishot resumptions. Naturally, not all embarrassingly parallel workloads exhibit

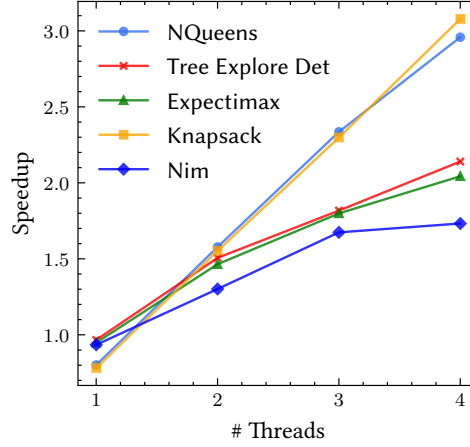


Figure 3.15: Strong-scaling behaviour of Zone LEXA on embarrassingly parallel multishot benchmarks. Speedups are relative to each benchmark’s sequential baseline.

perfect linear speedup; some show sublinear speedup due to unbalanced computation paths or extra sequential work after parallel execution.

3.7.3 Summary

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

3.8 Related Work

Lexical effect handlers. Lexically scoped handlers were designed to recover abstraction safety while retaining the expressiveness of effect handlers [69, 68, 4]. Subsequent work on continuation-passing-style (CPS) translation for Effekt made these semantics practical for real languages [45, 38]. It was demonstrated in LEXA that handler lookup can be made constant-time using stack switching [33] (Chapter 2), and Effekt adopted the same technique in its LLVM back end [37]. While these approaches all rely on passing handler references at run time, the associated overhead can sometimes outweigh that of handler

search, if effects are raised infrequently. Chapter 4 features an implementation strategy—for low-frequency effects—that does not rely on passing handler references [31]: it pays the cost of handler search in exchange for minimal overhead on mainline paths where effects are not raised.

Multi-prompt delimited control. Racket [18] supports multi-prompt delimited control, and LEXA and Effekt effectively support it too. Prompts, called *labels* in lexical effect handlers, are dynamically generated names. Techniques for finding the matching prompt in the evaluation context vary. In Racket, reducing the primitive `call-with-composable-continuation` involves locating the prompt by searching the stack. An earlier version of Effekt [38] uses a CPS translation that compiles labels away, but handler lookup effectively induces a search-like behaviour. Base LEXA [33] represents labels directly as stack addresses, which avoids the search but introduces pointer-invalidation issues when stacks are copied—the issue addressed in this chapter.

Support for multishot resumptions. Some systems support effect handlers but not multishot resumptions, citing pragmatic reasons. For example, in OCaml 5, resuming a continuation more than once is a run-time error [48], a choice motivated by the difficulty of reasoning about multishot continuations for both the programmer and the compiler. For lexically scoped handlers, supporting multishot resumptions is technically challenging because stack copying invalidates references to handlers installed on the copied stack. Effekt [37] recently introduced support for multishot resumptions by using a global pointer to route handler references to the active resumption copy; however, this approach still detects and aborts “reentrant resumptions”, precluding parallel execution of copies. Our virtual-resumptions technique removes this restriction while maintaining the fast lookup path expected of a stack-switching runtime, thereby enabling parallel execution of multishot resumptions.

Stack copying and address-space management. Implementation techniques for continuations have been studied extensively, and their trade-offs evaluated empirically [9, 16]. Beyond multishot resumptions, stack copying also arises when runtimes grow stacks by reallocation, as in Go [52] and OCaml [48]. Go’s runtime reallocates a stack by copying it to a new memory location and patching all pointers into the stack to the new location. OCaml’s runtime keeps a linked list of exception handlers on the stack, and the links of this list are likewise patched during reallocation.

3.9 Conclusion

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

Chapter 2 and this chapter both make the *handling* of effects fast, but they share a trait: each passes a run-time handler reference down the call chain, so even code that never raises an effect carries a small cost for the handlers in scope. That cost is a fair trade when effects are raised often, but for effect handlers whose effects are raised infrequently, it may be more efficient to pay the cost of handler search only when an effect is raised. The next chapter explores this design space.

Chapter 4

Zero LEXA: Zero-Overhead Lexical Effect Handlers

Chapters 2 and 3 pursued the efficient handling of effects: they made locating a handler, and capturing a continuation, constant-time operations by giving each handler a run-time identity (a stack address) that is passed down to the sites that raise effects to it. As the close of Chapter 3 observed, that run-time representation is not free: even code that never raises an effect carries it down the call chain. That cost is exactly what this chapter sets out to avoid. For a large class of programs that raise effects infrequently, the dominant concern is not the speed of raising an effect but the cost the mere presence of handlers imposes on the mainline, effect-free paths. This chapter presents *Zero LEXA*, a compilation strategy that satisfies the *zero-overhead principle*: code that does not raise an effect pays no run-time cost for having the handlers in scope. Zero LEXA reifies no representation of a handler and passes nothing extra down the call chain. This chapter is based on “Zero-Overhead Lexical Effect Handlers” [31], published at OOPSLA 2025.

4.1 The Zero-Overhead Principle for Lexical Handlers

Exception handling is a common feature of modern programming languages, designed to transfer control, in a nonlocal but structured manner, to handler code, separating common-case code from the handling of exceptional conditions. Effect handlers [41, 42] generalize exception handlers, letting programmers define custom delimited control operators. As Chapter 1 discussed, exception handlers (and effect handlers following them [2, 24, 27, 48])

have traditionally been *dynamically scoped*: when an effect is raised, the dynamically closest enclosing handler is chosen to handle it. A growing recognition that dynamic scoping compromises abstraction safety and modular reasoning [69, 68] has driven a wave of language designs toward *lexically scoped* handlers [69, 7, 33], in which a handler is a lexically scoped capability and only code that holds the capability can raise effects to it.

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

This is an instance of the *zero-overhead principle*.

“What you don’t use, you don’t pay for.”
— Stroustrup [49, 50]

In the context of exception handling, “zero-overhead exception handlers” means that code should not pay for exceptions unless they are actually raised. It does *not* mean that raising and handling exceptions should have no cost; on the contrary, it accepts a relatively high cost for the exceptional paths, which are expected to be unusual, in exchange for minimal overhead on the mainline paths.

“Normal case execution efficiency should not be impaired at all.”
— Liskov et al. [1, 29]

The principle has long shaped the design of exception handling, dating back to CLU [28], and has influenced C++, Java, and more recently WebAssembly [60] and Python [46]. In all of these languages, exception handlers are *dynamically* scoped, and zero-overhead strategies for dynamic scoping are well understood.

However, zero-overhead *lexically* scoped handlers remain elusive. The semantics of lexically scoped handlers involves generating fresh labels that identify installed handlers and passing these labels down to code that raises effects to them [69, 68, 4]. This semantics

suggests that implementations must represent these labels and propagate them at run time, incurring overhead even when no effects are raised. Indeed, all existing implementations do exactly that: handlers are identified via pooled objects [69], subregion evidence [45, 38], evidence vectors [65], or memory addresses [33] (Chapter 2), and extra instructions are needed to propagate these identifiers. Even when the actual overhead is small, the mere violation of the zero-overhead principle can discourage loss-averse compiler writers from adopting lexically scoped handlers over dynamic scoping.

This work. We present a compilation strategy for lexically scoped effect handlers that satisfies the zero-overhead principle. The key idea is a type-directed translation that lets the compiler emit static information tracking the lexical provenance of the handlers in scope. This information guides the runtime in walking the stack to locate the handler for a raised effect. The approach is complementary to, and compatible with, the strategies of Chapters 2 and 3: effects expected to be raised infrequently can use the new zero-overhead strategy, while effects expected to be raised frequently can still use the address-based strategy. A program can mix the two, letting the programmer choose a strategy per effect based on its expected usage.

4.2 Main Ideas

This section illustrates the main ideas of the zero-overhead strategy on a running example, the program in Figure 4.1. It consists of four pieces (a library, a plugin, a framework, and an application). It is similar to the example in Section 1.2, extended with a library function `libFun` used by the plugin. Lexical handlers and label passing work as introduced in Section 1.2; the code in pink marks what is new here—the *capability polymorphism* of `framework`—and can be ignored on a first reading. Although the code still shows the labels being passed as arguments in square brackets, we will soon see how our compilation strategy eliminates the need for passing them at run time yet still preserves the original semantics.

4.2.1 Handler Search Semantics and Implementation Strategies

Figure 4.2 shows the call stacks at the moment `libFun` raises a `Logging` effect during a call from the callback. Each column depicts how a `Logging` handler is found under a different semantics or compilation strategy. A large triangle ▲ represents the `consoleLogger`

```

1  effect Logging : string -> unit
2
3  def libFun [logger: Logging] (...) =
4    ...
5    raise logger "..."
6    ...
7
8  def plugin [logger: Logging] (x: int) =
9    ...
10   libFun [logger] (...)
11   ...
12
13  def framework [ $\alpha$ ] (callback: { $\alpha$ }int -> unit) =
14    handle (fileLogger: Logging) =>
15      callback (...)
16      raise fileLogger "..."
17    with  $\lambda$  msg k =>
18      writeToFile msg; resume k ()
19
20  def main () =
21    handle (consoleLogger: Logging) =>
22      framework [consoleLogger] (plugin [consoleLogger])
23    with  $\lambda$  msg k =>
24      writeToConsole msg; resume k ()

```

Figure 4.1: A program consisting of a library, a plugin, a framework, and an application. Code in pink pertains to capability polymorphism: `framework`'s capability variable α , its instantiation in `main`, and the capture set $\{\alpha\}$ in the type of `callback`.

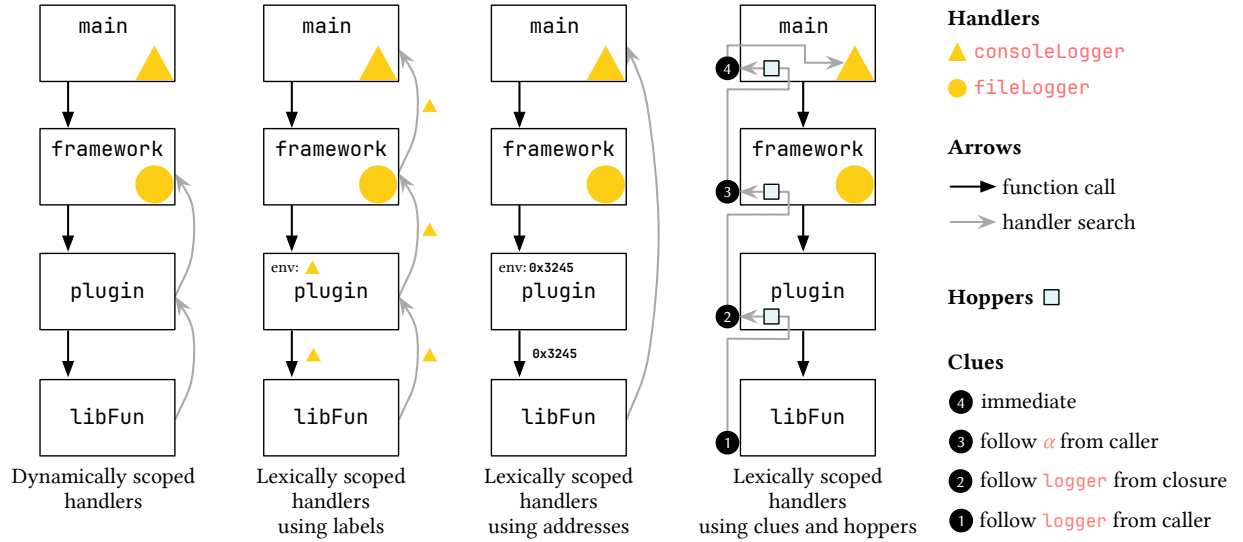


Figure 4.2: Stack diagrams illustrating different handler-search semantics and implementation strategies.

handler installed in `main`, and a large circle ● represents the `fileLogger` handler installed in `framework`; grey arrows depict the search behaviour.

The first column shows *dynamic scoping*: the runtime searches the stack for the nearest handler matching the effect name `Logging`, finds the one installed in `framework`, and logs the message to the file. This is the accidental handling introduced in Section 1.2: where the callback’s message ends up depends on whether the framework happens to install a `Logging` handler, so an implementation detail of the framework leaks.

The second column shows *lexical scoping*. When a handler is installed, a fresh label is generated to represent the new handler instance, and effects can be raised to it only through that label. In Figure 4.1, both `libFun` and `plugin` take a label `logger` for a `Logging` handler instance; `framework` is parameterized by a *capability variable* α abstracting over the set of labels its callback may capture. This polymorphism expresses that `framework` is oblivious to the callback’s effects. At run time, a fresh label (the small triangle ▲) is generated upon entering the `handle` block in `main` and substitutes for `consoleLogger`. `plugin` captures this label in its closure, and passes it to `libFun`, which raises to it. The runtime searches for the handler instance identified by this label, skips the one installed in `framework`, and reaches `consoleLogger` in `main`. Regardless of the framework’s implementation, the application is guaranteed that `plugin`’s messages go to its console.

The third column shows a representative efficient implementation of lexical handlers on

a low-level machine [33], the address-based strategy of Chapter 2. Instead of generating fresh labels, the compiler uses the stack address of the handler as the label, so a raise jumps directly to the handler. While raising is more efficient than the naive search of the previous column, there is still an overhead associated with passing the extra label argument, even when no effect is raised. Because control effects in practice (especially exceptions) are common in code but infrequent at run time, this violation of the zero-overhead principle is exactly what we wish to remove.

4.2.2 A Zero-Overhead Implementation Strategy

We propose an implementation strategy for lexically scoped handlers that meets the zero-overhead principle. Unlike existing approaches, it does not reify or pass any representation of handler instances at run time. Instead, the information needed to locate a handler is spread across the call stack. When an effect is raised, the runtime invokes a procedure called the *stackwalker*. The stackwalker always carries a piece of information called a *clue*. If the clue indicates that the stackwalker should walk further up the stack, it finds in the caller’s frame a transition function, called a *hopper*, that maps the current clue to the next clue, which it uses to continue the walk.

The fourth column of Figure 4.2 illustrates the stackwalker on the running example; each clue is shown as a textual description of what to do next (see the **Clues** legend in the figure).

1. When `libFun` raises the `Logging` effect, the initial clue tells the stackwalker to follow the `logger` label parameter of `libFun`, instantiated by its caller `plugin`.
2. At the call site of `libFun` in `plugin`, the hopper reveals that `plugin` called `libFun` with a `logger` label coming from `plugin`’s closure environment. Since the captured label must originate further up the stack, the walk continues.
3. At `framework`, the hopper reveals that the label captured in the callback is abstracted by the capability variable `α`; the stackwalker turns to follow `α`, which comes from the caller `main`, skipping the `fileLogger` installed in `framework`.
4. At `main`, the hopper reveals that `framework`’s capability variable `α` was instantiated by the label `consoleLogger`, whose handler lives in this very frame, so the stackwalker reaches the handler implementation directly.

```

1 effect F : unit -> unit
2
3 def fun0 [ $\alpha_0$ ,  $\ell_0$ : F,  $\ell_1$ : F]
4     (f: { $\ell_0, \alpha_0$ }unit -> unit) =
5     def fun1 { $\ell_1$ } [ $\alpha_1$ ,  $\ell_2$ : F]
6         (g: { $\alpha_1$ }unit -> unit) =
7         raise  $\ell_2$  ()
8         raise  $\ell_1$  ()
9         g ()
10    fun1 [( $\ell_0$ ,  $\alpha_0$ ),  $\ell_0$ ] (f)

```

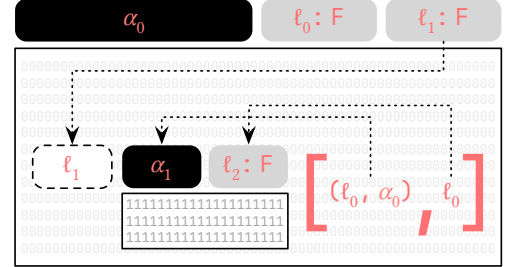


Figure 4.3: Higher-order functions with assorted binding structures. Above each function body, the black boxes are the capability-variable binders, the grey boxes the label-variable binders, and the dashed white boxes the function’s capture set; dotted arrows indicate the lexical provenance of each variable.

A hopper at a call site lets the stackwalker transition from walking the callee’s frame to walking the caller’s frame. To explain what information a hopper stores, we use the more elaborate example in Figure 4.3, where functions are parameterized by multiple label variables (ℓ) and capability variables (α). The function `fun0` is parameterized over a capability variable α_0 and two label variables ℓ_0 and ℓ_1 ; its body is a single call to a nested function `fun1`, which is parameterized over α_1 and a label variable ℓ_2 and captures ℓ_1 (shown in curly braces).

The right of Figure 4.3 depicts `fun0`’s definition, with `fun1`’s definition nested inside it. Above each function body (shaded with grey 0 or 1), the black boxes are the capability variables parameterizing the function, the grey boxes are its label parameters, and the dashed white boxes are its capture set (`fun1`’s capture set is $\{\ell_1\}$; `fun0`, a top-level function, captures nothing). The dotted arrows indicate the lexical provenance of the variables. The main enabler of the zero-overhead strategy is **a typing discipline that makes explicit which of the three kinds of binder (capture set, capability-variable parameter, or label-variable parameter) a handler in scope is associated with**. This discipline gives each call site enough knowledge, embedded in its hopper, about the provenance of a label or capability variable to send the stackwalker up the call stack: just reverse the arrows in Figure 4.3. Hoppers can be computed at compile time, and the stackwalker needs only the return address of a call site to locate its hopper (Section 4.5); nothing beyond the return address, already on the stack, needs to be stored.

Figure 4.4 shows three stack snapshots illustrating how hoppers guide the stackwalker,

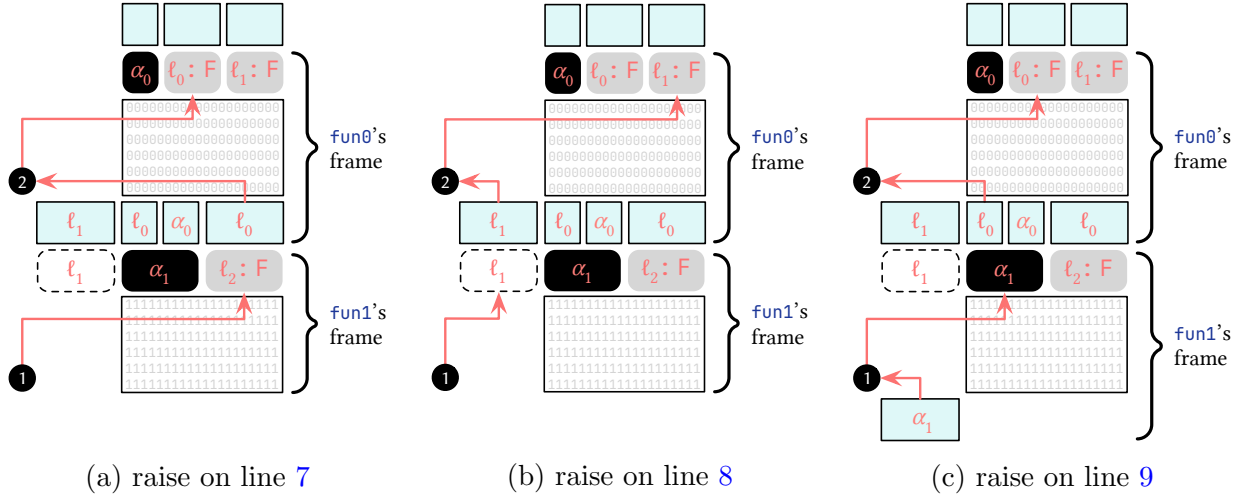


Figure 4.4: Stack snapshots illustrating the stackwalker’s behaviour when effects are raised from different points of the program in Figure 4.3. Clues are shown as numbered circles.

each for a different effect raised from `fun1` or below it. Frames are shaded grey 0 or grey 1 for `fun0` or `fun1`. Between two frames sit the three kinds of binder, and above each binder box sit one or more light-blue boxes; this is where the hopper conceptually resides. It records how each parameter of the callee is instantiated and the provenance of the captured variables.

- In snapshot (a), `fun1` raises ℓ_2 (line 7). The stackwalker starts with an initial clue ❶ that the handler traces back to the label parameter ℓ_2 of `fun1`, so it queries the hopper in the caller’s frame. Given clue ❶, the hopper returns a new clue ❷: the handler traces back to the label parameter ℓ_0 of `fun0`. The walk continues.
- In snapshot (b), `fun1` raises ℓ_1 (line 8), a free variable *captured* by `fun1`. By the typing discipline, captured variables are reflected in a function’s type, and the compiler includes their provenance in the hopper at the call site. So, given the initial clue ❶, the hopper returns clue ❷: trace back to the label parameter ℓ_1 of `fun0`.
- In snapshot (c), the effect is raised from below `fun1`’s frame, by its callee `f` (passed as `g`, line 9). Queried at the call site of `g` with a clue that the handler comes from `g`’s capture set, the hopper, which records the provenance of the capture set, returns clue ❶: trace back to the capability variable α_1 of `fun1`. Using clue ❶ at the call site of `fun1`, the hopper returns clue ❷: trace back to the label parameter ℓ_0 of `fun0`.

At the last hopper, how does the stackwalker know that it should follow ℓ_0 and not α_0 ? The intended handler might have been abstracted by α_0 . We resolve this potential ambiguity next.

4.2.3 Non-Ambiguity Requirement on Capture Sets

In snapshot (c) of Figure 4.4, at the call site of `fun1`, the stackwalker encounters the instantiation of α_1 and faces two candidates: the handler may be traced back to the label parameter ℓ_0 of `fun0`, or it may hide behind the capability variable α_0 . We resolve the ambiguity by combining a run-time mechanism with a static restriction.

The run-time mechanism is to make clues carry the name of the raised effect. Suppose the raised effect is F ; the stackwalker carries F as part of the clue, and it queries the hopper at the call site of `fun1` and finds that, in the instantiation (ℓ_0, α_0) , there is exactly one label ℓ_0 declared with effect name F , so it chooses to follow ℓ_0 .

A matching effect name alone is not conclusive, however: there could be multiple matches, or the intended handler could still be hidden behind α_0 , whose contents are statically unknown. The static restriction eliminates this possibility at its source: for every function *definition*, a nonempty capture set must consist of either a single capability variable or labels with distinct effect names. Under this restriction, a label with a matching effect name is guaranteed to be the intended handler, so the stackwalker need not look further; if no label matches, the handler must be behind the capability variable, of which there is at most one. The restriction is enforced by the type system (Section 4.3) and lets hopper uniquely determine the label or capability variable to follow.

This non-ambiguity requirement is mild. Existing surface-language designs for lexically scoped handlers enforce a condition stronger than ours [69, 7]: if two effects with the same effect name are raised from the same function, they must be handled by the same handler. The stronger condition buys a lightweight surface syntax with no explicit parameterization over capability and label variables; our implementation strategy applies readily to such surface languages.

Conversely, it is sometimes useful to let multiple handler instances of the same effect name coexist in the same scope [65], and our strategy supports this expressiveness. We emphasize that the non-ambiguity requirement applies to the capture set of a function, not to its parameters. Therefore, a function may freely be parameterized by multiple handlers of the same effect name: `fun0` in Figure 4.3, for instance, takes both ℓ_0 and ℓ_1 of effect name F . The one place where the non-ambiguity requirement restricts expressiveness is in handled code blocks. In `handle C with H`, the block `C` must be typed like a function

definition, because once H handles an effect raised within C , the remaining computation of C may be reified as a continuation, which behaves like a function. The labels that C captures must therefore have distinct effect names, which limits how many same-effect handlers coming from an outer scope can be used in C . This limitation vanishes when H is tail-resumptive or abortive, since such handlers never reify the continuation: it is either resumed in place or discarded. Because our strategy chiefly targets exceptional effects, which are typically abortive, the practical impact of the non-ambiguity restriction is minimal.

4.3 A Formal Model of the Compilation Scheme

We capture the essence of the compilation scheme as a type-directed translation between two core languages. In the source language SL , handlers are identified by freshly generated labels passed down to effect-raising sites; in the target language TL , handlers are located purely by piecing together information left on the evaluation context. Everything else is, to some extent, an implementation detail. Both languages are based on the simply-typed lambda calculus, and their semantics are given as abstract machines. SL is intentionally not high-level enough to be a surface language, and TL not low-level enough to be a machine language: the aim is to make the semantic gap between them concern only how handlers are located. This section presents the languages and the type-directed translation, and Section 4.4 develops the metatheory; the full proofs appear in Appendix C.

4.3.1 The Source Language SL

Figure 4.5 gives the syntax of SL ; the type system is deferred to Section 4.3.3. An overline denotes a (possibly empty) sequence of syntactic elements, with the empty sequence written ϵ ; the i -th element of a sequence is written $\bar{\bullet}^{(i)}$, and capture-avoiding substitution is written $\bullet[\bullet \mapsto \bullet]$. Values include variables, the unit value, and abstractions; an abstraction $[\bar{\alpha}; \bar{\ell}](\bar{x}) \Rightarrow t$ may be parameterized by a sequence $\bar{\alpha}$ of *capability variables* and a sequence $\bar{\ell}$ of *label variables*. A *label* l substitutes for a label variable ℓ , and a *capability* T (a sequence of labels and at most one capability variable) substitutes for a capability variable α . Parameterizing over $\bar{\alpha}$ lets a higher-order function be polymorphic over the capabilities its function arguments require. That a capability may contain at most one capability variable is a simplification rather than a limitation; see the end of Section 4.3.3 for a discussion.

In an application $v_1[\bar{T}; \bar{l}](\bar{v}_2)$, the function v_1 receives the capabilities $\bar{T}$ instantiating its capability variables, the labels $\bar{l}$ instantiating its label variables, and the arguments $\bar{v}_2$. The **handle** construct **handle** $[\ell : F] \Rightarrow t_1$ **with** $(x, k) \Rightarrow t_2$ binds the label variable ℓ in the handled computation t_1 ; the handler t_2 binds the payload x of the raised effect and the resumption k . A **raise** expression **raise** $l(v)$ raises an effect to the handler identified by the label l , passing v as the payload. A term sequences expressions with **let** bindings. SL programs are in A-normal form [17]: every subexpression, except those in the **handle** and **with** clauses, is a value. We make the standard simplifications that an effect signature contains exactly one effect operation and that an effect operation takes exactly one argument.

Figure 4.7 gives the operational semantics as an abstract machine, with the runtime syntax in Figure 4.6. The syntax of labels l is extended with a metavariable L ranging over labels freshly generated at run time, which do not appear in program text. The machine state M is in one of three modes: a *normal mode* $\langle E \parallel t \rangle$ with an evaluation context E and a redex t ; a *search mode* $\langle E \parallel K \parallel L \parallel v \rangle$ with a continuation K under construction, a label L , and a payload v ; and a *rewind mode* $\langle E \parallel K \parallel v \rangle$ with a continuation K to be rewound. An evaluation context is a sequence of frames: a *let frame* **let** $x = \square$ **in** t or a *handler frame* **let** $x = \#L^{(y,k) \Rightarrow t_2} \square$ **in** t_1 , which records a freshly generated label L identifying the handler instance together with the handler code. A continuation is like an evaluation context but is constructed inside-out rather than outside-in. The syntax of values is extended with reified continuations **cont** $\#L^{(x,k) \Rightarrow t} K$. SL uses the standard deep-handler semantics [23], so every captured continuation begins with a handler frame.

Because SL programs are in A-normal form, the syntax already determines the order of evaluation, and no structural rules are needed to locate the next redex. The rules S-APP, S-RETURN, S-LEAVE, and S-DONE are straightforward; in S-APP, the capabilities and labels supplied at the call site substitute for the callee’s capability and label variables. The rule S-HANDLE installs a handler frame carrying a freshly generated label L that identifies the newly installed handler instance, and the handled computation t_1 becomes the next redex. This generativity of labels at handler installation is shared by all prior calculi for lexically scoped handlers [69, 68, 4]. The rule S-RAISE suspends the current computation: the machine transitions to search mode, seeking the handler frame that matches the label L given as the first operand of **raise**. In search mode, the unwinding rules S-UNW-LET and S-UNW-HDL move frames off the evaluation context and accumulate them into the continuation under construction; when the handler frame matching L is reached, S-FOUND transitions back to normal mode, with the handler code t_2 as the next redex, the accumulated continuation substituting for the resumption k , and the payload v for y . The rule S-RESUME restores a captured continuation: the machine enters rewind

mode, where S-REW-LET and S-REW-HDL move the frames of the continuation back onto the evaluation context; when no frames remain, S-DONE returns the machine to normal mode with the payload in redex position.

effect name F label variable ℓ capability variable α term variable x, y, z, k

label $l ::= \ell$

value $v ::= x \mid () \mid [\bar{\alpha}; \bar{\ell}] (\bar{x}) \Rightarrow t$

capability $T ::= \epsilon \mid \alpha \mid T, l$

expression $e ::= v_1 [\bar{T}; \bar{l}] (\bar{v}_2) \mid \mathbf{handle} [\ell:F] \Rightarrow t_1 \mathbf{with} (x, k) \Rightarrow t_2 \mid \mathbf{raise} l(v) \mid \mathbf{resume} v_1(v_2)$

term $t ::= v \mid \mathbf{let} x = e \mathbf{in} t$

Figure 4.5: Syntax of SL.

label $l ::= \dots \mid L$

value $v ::= \dots \mid \mathbf{cont} \#L^{(x,k) \Rightarrow t} K$

frame $A ::= \mathbf{let} x = \square \mathbf{in} t \mid \mathbf{let} x = \#L^{(y,k) \Rightarrow t_2} \square \mathbf{in} t_1$

evaluation context $E ::= \epsilon \mid E \cdot A$

continuation $K ::= \epsilon \mid A \cdot K$

configuration $M ::= \langle E \parallel t \rangle \mid \langle E \parallel K \parallel L \parallel v \rangle \mid \langle E \parallel K \parallel v \rangle$

Figure 4.6: Runtime syntax of SL.

$$\begin{array}{ll}
\text{S-APP} & \langle E \parallel \text{let } y = ([\bar{\alpha}; \bar{\ell}] (\bar{x}) \Rightarrow t_1) [\bar{T}; \bar{L}] (\bar{v}) \text{ in } t_2 \rangle \longrightarrow \\
& \langle E \cdot (\text{let } y = \square \text{ in } t_2) \parallel t_1 [\bar{\alpha} \mapsto \bar{T}, \bar{\ell} \mapsto \bar{L}, \bar{x} \mapsto \bar{v}] \rangle \\
\text{S-RETURN} & \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel v \rangle \longrightarrow \langle E \parallel t [x \mapsto v] \rangle \\
\text{S-HANDLE} & \langle E \parallel \text{let } x = (\text{handle } [\ell:F] \Rightarrow t_1 \text{ with } (y,k) \Rightarrow t_2) \text{ in } t_3 \rangle \longrightarrow \\
& \langle E \cdot (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_3) \parallel t_1 [\ell \mapsto L] \rangle \quad \text{where } L \text{ is fresh} \\
\text{S-LEAVE} & \langle E \cdot (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \parallel v \rangle \longrightarrow \langle E \parallel t_1 [x \mapsto v] \rangle \\
\\
\text{S-RAISE} & \langle E \parallel \text{let } x = \text{raise } L(v) \text{ in } t \rangle \longrightarrow \langle E \parallel \text{let } x = \square \text{ in } t \parallel L \parallel v \rangle \\
\text{S-UNW-LET} & \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel L \parallel v \rangle \longrightarrow \langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel L \parallel v \rangle \\
\text{S-UNW-HDL} & \langle E \cdot (\text{let } x = (\#L'^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \parallel K \parallel L \parallel v \rangle \longrightarrow \\
& \langle E \parallel (\text{let } x = (\#L'^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \cdot K \parallel L \parallel v \rangle \quad \text{where } L \neq L' \\
\text{S-FOUND} & \langle E \cdot (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \parallel K \parallel L \parallel v \rangle \longrightarrow \\
& \langle E \cdot (\text{let } x = \square \text{ in } t_1) \parallel t_2 [y \mapsto v, k \mapsto \text{cont } \#L^{(y,k) \Rightarrow t_2} K] \rangle \\
\\
\text{S-RESUME} & \langle E \parallel \text{let } x = \text{resume } (\text{cont } \#L^{(y,k) \Rightarrow t_2} K) (v) \text{ in } t_1 \rangle \longrightarrow \\
& \langle E \cdot (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \parallel K \parallel v \rangle \\
\text{S-REW-LET} & \langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel v \rangle \longrightarrow \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel v \rangle \\
\text{S-REW-HDL} & \langle E \parallel (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \cdot K \parallel v \rangle \longrightarrow \\
& \langle E \cdot (\text{let } x = (\#L^{(y,k) \Rightarrow t_2} \square) \text{ in } t_1) \parallel K \parallel v \rangle \\
\text{S-DONE} & \langle E \parallel \epsilon \parallel v \rangle \longrightarrow \langle E \parallel v \rangle
\end{array}$$

Figure 4.7: Operational semantics of **SL** as an abstract machine.

4.3.2 The Target Language **TL**

TL (Figure 4.8) looks similar to **SL**, with one crucial difference: no label variables or capability variables appear in it. Abstractions are not parameterized by them, the **handle** construct binds no label, and, most importantly, **raise** does not reference any label but instead carries a *clue*. A clue consists of an *index* and the name of the effect being raised

(as in a language with dynamically scoped handlers); the index takes one of three forms: $\hat{i}$, $\hat{\circ}i$, or ∞ . The index $\hat{i}$, called a *label index*, is the TL counterpart of an SL label variable. It corresponds to the label variable at de Bruijn [12] index i . Similarly, $\hat{\circ}i$, called a *capability index*, corresponds to the capability variable at index i , and ∞ refers to a variable in the capture set. TL applications, **handle** expressions, and **resume** expressions are enriched with *call-site metadata* (H or T), produced during the translation from SL to TL. Hoppers are defined in terms of this metadata: when an effect is raised, the runtime uses the hopper at each call site in the evaluation context to update the clue until the right handler is found.

The operational semantics of TL (Figure 4.9) is given as an abstract machine. An evaluation context is built from three kinds of frames: plain let frames, *call frames* **let** $x = \square^H$ **in** t , which carry the call-site metadata H , and handler frames, which carry the metadata T . Unlike in SL, handler frames are not marked by labels, as no labels exist in TL. The machine has the same three modes as SL's, but in search mode it carries a clue C rather than a label. In normal mode, T-APP and T-HANDLE install a call frame and a handler frame, respectively, attaching the metadata H or T to the new frame, and T-RET, T-RET-APP, and T-LEAVE pop frames when a term finishes evaluating.

The rule T-RAISE transitions the machine from normal mode to search mode, with the **raise** expression providing the initial clue; the unwinding rules T-UNW-LET, T-UNW-APP, and T-UNW-HDL then take over, moving frames from the evaluation context onto the continuation under construction. A plain let frame is skipped with the clue unchanged (T-UNW-LET). The rule T-UNW-APP skips over a call frame, updating the clue using the call-site metadata H : the new clue is $\text{hopper}_H(C)$, where hopper_H is a meta-level transition function whose definition is the subject of Section 4.3.3. The T-FOUND rule fires when the enclosing frame is a handler frame and the clue index is $\hat{\circ}$, meaning that this handler should handle the effect: the handler body runs with the payload bound to x and the resumption—the continuation accumulated during the search, delimited by the handler—bound to k . Otherwise T-UNW-HDL skips the handler frame and continues, updating the clue with $\text{hopper}_{T;\epsilon;\epsilon}$, where T is the metadata carried by the skipped frame. Finally, T-RESUME enters rewind mode, and the rules T-REW-LET, T-REW-APP, and T-REW-HDL replay the frames of the resumed continuation back onto the evaluation context, after which T-DONE returns the machine to normal mode.

label index	ℓ	$::= \infty \mid \hat{i}$
capability index	α	$::= \infty \mid \overset{\circ}{i}$
value	v	$::= x \mid () \mid (\bar{x}) \Rightarrow t \mid \mathbf{cont} \#^{(x,k) \Rightarrow t} K$
clue	C	$::= \langle \ell, F \rangle \mid \langle \alpha, F \rangle$
capability	T	$::= \epsilon \mid \alpha \mid T, \ell$
call-site metadata	H	$::= T_0; \bar{T}; \bar{\ell}:\bar{F}$
expression	e	$::= v (\bar{v})^H \mid \mathbf{handle}^T t_1 \mathbf{with} (x, k) \Rightarrow t_2 \mid \mathbf{raise} C(v) \mid \mathbf{resume}^T v(v)$
term	t	$::= v \mid \mathbf{let} x = e \mathbf{in} t$
frame	A	$::= \mathbf{let} x = \square \mathbf{in} t \mid \mathbf{let} x = \square^H \mathbf{in} t \mid \mathbf{let} x = (\#^{(y,k) \Rightarrow t} \square)^T \mathbf{in} t$
evaluation context	E	$::= \epsilon \mid E \cdot A$
continuation	K	$::= \epsilon \mid A \cdot K$
configuration	M	$::= \langle E \parallel t \rangle \mid \langle E \parallel K \parallel C \parallel v \rangle \mid \langle E \parallel K \parallel v \rangle$

Figure 4.8: Syntax of **TL**, including its runtime forms.

$$\begin{array}{lcl}
\text{T-APP} & \langle E \parallel \text{let } x = ((\bar{y}) \Rightarrow t_1) (\bar{v})^H \text{ in } t_2 \rangle \longrightarrow & \\
& \langle E \cdot (\text{let } x = \square^H \text{ in } t_2) \parallel t_1 [\bar{y} \mapsto \bar{v}] \rangle & \\
\text{T-RET} & \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel v \rangle \longrightarrow \langle E \parallel t[x \mapsto v] \rangle & \\
\text{T-RET-APP} & \langle E \cdot (\text{let } x = \square^H \text{ in } t) \parallel v \rangle \longrightarrow \langle E \parallel t[x \mapsto v] \rangle & \\
\text{T-HANDLE} & \langle E \parallel \text{let } x = (\text{handle}^T t_1 \text{ with } (y, k) \Rightarrow t_2) \text{ in } t_3 \rangle \longrightarrow & \\
& \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_3) \parallel t_1 \rangle & \\
\text{T-LEAVE} & \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \parallel v \rangle \longrightarrow \langle E \parallel t_1[x \mapsto v] \rangle & \\
\\
\text{T-RAISE} & \langle E \parallel \text{let } x = \text{raise } C(v) \text{ in } t \rangle \longrightarrow \langle E \parallel \text{let } x = \square \text{ in } t \parallel C \parallel v \rangle & \\
\text{T-UNW-LET} & \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel C \parallel v \rangle \longrightarrow \langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel C \parallel v \rangle & \\
\text{T-UNW-APP} & \langle E \cdot (\text{let } x = \square^H \text{ in } t) \parallel K \parallel C \parallel v \rangle \longrightarrow & \\
& \langle E \parallel (\text{let } x = \square^H \text{ in } t) \cdot K \parallel \text{hopper}_H(C) \parallel v \rangle & \\
\text{T-UNW-HDL} & \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \parallel K \parallel C \parallel v \rangle \longrightarrow & \\
& \langle E \parallel (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \cdot K \parallel \text{hopper}_{T;\epsilon;\epsilon}(C) \parallel v \rangle & \\
& \text{where } C \neq \langle \hat{0}, F \rangle \text{ for any } F & \\
\text{T-FOUND} & \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \parallel K \parallel \langle \hat{0}, F \rangle \parallel v \rangle \longrightarrow & \\
& \langle E \cdot (\text{let } x = \square^{T;\epsilon;\epsilon} \text{ in } t_1) \parallel t_2 [y \mapsto v, k \mapsto \text{cont } \#^{(y,k) \Rightarrow t_2} K] \rangle & \\
\\
\text{T-RESUME} & \langle E \parallel \text{let } x = \text{resume}^T (\text{cont } \#^{(y,k) \Rightarrow t_2} K) (v) \text{ in } t_1 \rangle \longrightarrow & \\
& \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \parallel K \parallel v \rangle & \\
\text{T-REW-LET} & \langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel v \rangle \longrightarrow \langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel v \rangle & \\
\text{T-REW-APP} & \langle E \parallel (\text{let } x = \square^H \text{ in } t) \cdot K \parallel v \rangle \longrightarrow \langle E \cdot (\text{let } x = \square^H \text{ in } t) \parallel K \parallel v \rangle & \\
\text{T-REW-HDL} & \langle E \parallel (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \cdot K \parallel v \rangle \longrightarrow & \\
& \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_1) \parallel K \parallel v \rangle & \\
\text{T-DONE} & \langle E \parallel \epsilon \parallel v \rangle \longrightarrow \langle E \parallel v \rangle &
\end{array}$$

Figure 4.9: Operational semantics of TL as an abstract machine.

4.3.3 The Type-Directed Translation

The translation from SL to TL is type-directed: Figure 4.10 gives the type-level constructs and the auxiliary non-ambiguity and non-frivolity judgements, and Figures 4.11 and 4.12 give the variable-translation judgements and selected typing rules, with the translation output shown in orange.

The SL type system. A type is the unit type, a function type, or a continuation type; function and continuation types are annotated with a capture set T recording the capability or label variables captured by the function or continuation. Expressions are type-checked under four contexts: Δ and Σ for the capability and label variables *parameterizing* the immediately enclosing abstraction, Θ for the variables *lexically captured* by it, and Γ for term variables. The rules are implicitly parameterized by a global context $\mathbb{F}$ mapping effect names to effect signatures. Using three contexts Θ , Δ , Σ makes explicit which of the three kinds of binder (cf. Figure 4.3) a handler in scope is associated with; this distinction is what the stackwalker needs, and it neither restricts nor relaxes which SL programs are typable.

The rules ST-UNIT, ST-VAR, and ST-LET are straightforward. In ST-FUN, the function body is typed under the capture set T , whose contents may come from any of Θ , Δ , and Σ . The rule requires T to be *unambiguous*: the judgement $\Theta \mid \Delta \mid \Sigma \vdash T$ as Θ' unamb checks the non-ambiguity condition of Section 4.2.3—a nonempty T consists of either a single capability variable or labels with distinct effect names—and produces the capture context Θ' under which the body is typed. The rule also requires the function type to be *non-frivolous*: each capability variable the function is parameterized over must appear in the capture set of some argument. This condition is a sanity check that the capability variables stand for capabilities needed by the function arguments.

In ST-HANDLE, the **handle** expression can be viewed as defining two functions—the handled computation t_1 , parameterized by the label variable ℓ , and the handler t_2 , parameterized by the payload x and the resumption k —and then applying the first. Both are typed under the same capture set T , which must be unambiguous so that it can serve as their capture context Θ' ; the continuation type of k carries the same capture set. The handled computation t_1 is typed under a label context containing the single label $\ell:F$, which serves as the lexical identifier of the handler. A standard escaping condition—the answer type τ_{ans} must be well-formed without ℓ —prevents the label from escaping its scope, as in type-and-effect systems for lexical handlers [68, 4] or lexical regions [57, 10, 19]. The remaining rules are standard: ST-APP types each argument at the corresponding parameter type with the callee’s label and capability variables instantiated, ST-RAISE uses the effect

signature to type-check the payload, and ST-RESUME requires the input type of the continuation to match the payload and gives the `resume` expression the continuation’s answer type.

Compared with prior type systems for lexically scoped handlers, SL’s differs in two ways. First, it separates function parameters (Δ, Σ) from captured variables (Θ) ; as noted above, this changes nothing about typability. Second, it imposes the non-ambiguity condition on the capture set of every function definition and handled computation, which in theory reduces expressiveness but is of little practical concern (Section 4.2.3). Because SL’s typing conditions strengthen those of prior type systems shown to be sound, we omit a type-soundness proof.

Call-site metadata. The TL program produced from a well-typed SL program uses call-site metadata to track the lexical provenance of the label and capability variables that the SL callee has access to. Call-site metadata $H = (T_0; \bar{T}; \bar{\ell}:\bar{F})$ is a three-tuple, where T_0 is the callee’s capture set, $\bar{T}$ instantiates its capability variables, and $\bar{\ell}:\bar{F}$ instantiates its label variables. Call-site metadata is a TL concept: all three components are expressed with TL indices rather than SL labels or capabilities.

The translation. The typing rules simultaneously perform the translation. The variable-translation judgements $\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \ell$ and $\Theta \mid \Delta \vdash \alpha \rightsquigarrow \alpha$ map SL variables to TL indices: a label variable bound by the immediately enclosing abstraction (one in Σ) is translated to the label index $\hat{i}$ given by its de Bruijn position, a capability variable (one in Δ) to the capability index i , and a captured variable of either kind to ∞ . Capabilities and call-site metadata are translated structurally from these judgements (the rules are omitted from the figures). The interesting rules are the four that emit metadata or clues. In ST-APP, the metadata H is computed from the callee’s capture set T and the instantiation $[\bar{T}_1; \bar{\ell}_1]$ at the call site. In ST-HANDLE and ST-RESUME, the construct is annotated with the translation T of the capture set, since entering a handled computation or resuming a continuation also transfers control to code whose handlers’ provenance the stackwalker must be able to follow. In ST-RAISE, the label variable ℓ is translated to the initial clue $\langle \ell, F \rangle$: the TL index of ℓ paired with the name of the raised effect.

type $\tau ::= \mathbf{unit} \mid \{T\} \forall [\bar{\alpha}; \bar{\ell}:F] (\bar{\tau}_1) \rightarrow \tau_2 \mid \{T\} \mathbf{cont} \tau_1 \rightarrow \tau_2$
 capture context $\Theta ::= \epsilon \mid \alpha \mid \bar{\ell}:F$
 capability context $\Delta ::= \epsilon \mid \Delta, \alpha$
 label context $\Sigma ::= \epsilon \mid \Sigma, \ell:F$
 term context $\Gamma ::= \epsilon \mid \Gamma, x:\tau$
 effect signatures $\mathbb{F} ::= \epsilon \mid \mathbb{F}, F:\tau \rightarrow \tau$

$$\boxed{\Theta \mid \Delta \mid \Sigma \vdash T \text{ as } \Theta' \text{ unamb}}$$

$$\Theta \mid \Delta \mid \Sigma \vdash \epsilon \text{ as } \epsilon \text{ unamb} \quad \frac{\Theta \mid \Delta \vdash \alpha}{\Theta \mid \Delta \mid \Sigma \vdash \alpha \text{ as } \alpha \text{ unamb}}$$

$$\frac{\Theta \mid \Delta \mid \Sigma \vdash T \text{ as } \bar{\ell}':\bar{F}' \text{ unamb} \quad \Theta \mid \Sigma \vdash \ell:F \quad F \notin \bar{F}'}{\Theta \mid \Delta \mid \Sigma \vdash T, \ell \text{ as } \bar{\ell}':\bar{F}', \ell:F \text{ unamb}}$$

$$\boxed{\vdash \{T\} \forall [\bar{\alpha}; \bar{\ell}:F] (\bar{\tau}_1) \rightarrow \tau_2 \text{ nonfriv}}$$

$$\frac{\forall \alpha_0 \in \bar{\alpha}. \exists i. \bar{\tau}_1^{(i)} = \{T'\} \forall [\bar{\alpha}'; \bar{\ell}':F'] (\bar{\tau}_1') \rightarrow \tau_2' \wedge \alpha_0 \in T'}{\vdash \{T\} \forall [\bar{\alpha}; \bar{\ell}:F] (\bar{\tau}_1) \rightarrow \tau_2 \text{ nonfriv}}$$

Figure 4.10: Type-level constructs and auxiliary judgements of SL.

$$\begin{array}{c}
\boxed{\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \ell} \qquad \boxed{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \alpha} \\
\\
\frac{\text{index}_\Sigma(\ell) = i \quad \Sigma(\ell) = F}{\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \hat{i}} \qquad \frac{\text{index}_\Delta(\alpha) = i}{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \hat{i}} \\
\\
\frac{\ell : F \in \Theta}{\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \infty} \qquad \frac{\alpha \in \Theta}{\Theta \mid \Delta \vdash \alpha \rightsquigarrow \infty} \\
\\
\boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v : \tau \rightsquigarrow v} \qquad \boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash e : \tau \rightsquigarrow e} \qquad \boxed{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash t : \tau \rightsquigarrow t} \\
\\
\text{ST-UNIT} \quad \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash () : \mathbf{unit} \rightsquigarrow () \qquad \text{ST-VAR} \quad \frac{\Gamma(x) = \tau}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash x : \tau \rightsquigarrow x} \\
\\
\text{ST-LET} \quad \frac{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash e : \tau_1 \rightsquigarrow e \quad \Theta \mid \Delta \mid \Sigma \mid \Gamma, x : \tau_1 \vdash t : \tau_2 \rightsquigarrow t}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \mathbf{let } x = e \mathbf{ in } t : \tau_2 \rightsquigarrow \mathbf{let } x = e \mathbf{ in } t} \\
\\
\text{ST-FUN} \quad \frac{\Theta' \mid \bar{\alpha} \mid \bar{\ell} : \bar{F} \mid \Gamma, \bar{x} : \bar{\tau}_1 \vdash t : \tau_2 \rightsquigarrow t \quad \Theta \mid \Delta \mid \Sigma \vdash T \text{ as } \Theta' \text{ unamb} \quad \vdash \{T\} \forall [\bar{\alpha}; \bar{\ell} : \bar{F}] (\bar{\tau}_1) \rightarrow \tau_2 \text{ nonfriv}}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash [\bar{\alpha}; \bar{\ell}] (\bar{x}) \Rightarrow t : \{T\} \forall [\bar{\alpha}; \bar{\ell} : \bar{F}] (\bar{\tau}_1) \rightarrow \tau_2 \rightsquigarrow (\bar{x}) \Rightarrow t} \\
\\
\text{ST-APP} \quad \frac{\begin{array}{c} \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1 : \{T\} \forall [\bar{\alpha}; \bar{\ell} : \bar{F}] (\bar{\tau}) \rightarrow \tau_1 \rightsquigarrow v_1 \\ \forall i. \Theta \mid \Delta \mid \Sigma \vdash \bar{T}_1^{(i)} \quad \forall i. \Theta \mid \Sigma \vdash \bar{\ell}_1^{(i)} : \bar{F}^{(i)} \\ \forall i. \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \bar{v}_2^{(i)} : \bar{\tau}^{(i)} [\bar{\ell} \mapsto \bar{\ell}_1] [\bar{\alpha} \mapsto \bar{T}_1] \rightsquigarrow \bar{v}_2^{(i)} \quad \Theta \mid \Delta \mid \Sigma \vdash T; \bar{T}_1; \bar{\ell}_1 : \bar{F} \rightsquigarrow H \end{array}}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1 [\bar{T}_1; \bar{\ell}_1] (\bar{v}_2) : \tau_1 [\bar{\ell} \mapsto \bar{\ell}_1] [\bar{\alpha} \mapsto \bar{T}_1] \rightsquigarrow v_1 (\bar{v}_2)^H}
\end{array}$$

Figure 4.11: The SL type system and type-directed translation: selected rules (part 1).

$$\begin{array}{c}
\text{ST-HANDLE} \quad \frac{\mathbb{F}(F) = (\tau_1) \rightarrow \tau_2 \quad \Theta \mid \Delta \mid \Sigma \vdash T \text{ as } \Theta' \text{ unamb} \quad \Theta \mid \Delta \mid \Sigma \vdash T \rightsquigarrow T \\
\Theta \mid \Delta \mid \Sigma \vdash \tau_{\text{ans}} \quad \Theta' \mid \epsilon \mid \ell : F \mid \Gamma \vdash t_1 : \tau_{\text{ans}} \rightsquigarrow t_1 \\
\Theta' \mid \epsilon \mid \epsilon \mid \Gamma, x : \tau_1, k : \{T\} \text{ cont } \tau_2 \rightarrow \tau_{\text{ans}} \vdash t_2 : \tau_{\text{ans}} \rightsquigarrow t_2}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \text{handle}[\ell : F] \Rightarrow t_1 \text{ with } (x, k) \Rightarrow t_2 : \tau_{\text{ans}} \rightsquigarrow \text{handle}^T t_1 \text{ with } (x, k) \Rightarrow t_2} \\
\\
\text{ST-RAISE} \quad \frac{\Theta \mid \Sigma \vdash \ell : F \rightsquigarrow \ell \quad \mathbb{F}(F) = \tau_1 \rightarrow \tau_2 \quad \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v : \tau_1 \rightsquigarrow v}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \text{raise } \ell(v) : \tau_2 \rightsquigarrow \text{raise } \langle \ell, F \rangle (v)} \\
\\
\text{ST-RESUME} \quad \frac{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_1 : \{T\} \text{ cont } \tau_1 \rightarrow \tau_2 \rightsquigarrow v_1 \quad \Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash v_2 : \tau_1 \rightsquigarrow v_2 \\
\Theta \mid \Delta \mid \Sigma \vdash T \rightsquigarrow T}{\Theta \mid \Delta \mid \Sigma \mid \Gamma \vdash \text{resume } v_1(v_2) : \tau_2 \rightsquigarrow \text{resume}^T v_1(v_2)}
\end{array}$$

Figure 4.12: The SL type system and type-directed translation: selected rules (part 2).

$$\boxed{\text{hopper}_H(C) = C'} \quad \text{where } H = T_0; \bar{T}; \bar{\ell} : \bar{F}$$

$$\begin{aligned}
\text{hopper}_H(\langle \hat{i}, F' \rangle) &= \langle \ell', F' \rangle \quad \text{where } \ell' : F' = \bar{\ell} : \bar{F}^{(i)} \\
\text{hopper}_H(\langle \overset{\circ}{i}, F' \rangle) &= \begin{cases} \langle \ell', F' \rangle & \text{if there is a unique } \ell' \text{ such that } \ell' : F' \in \bar{T}^{(i)} \\ \langle \alpha, F' \rangle & \text{else if there is an } \alpha \text{ such that } \alpha \in \bar{T}^{(i)} \end{cases} \\
\text{hopper}_H(\langle \infty, F' \rangle) &= \begin{cases} \langle \ell', F' \rangle & \text{if there is a unique } \ell' \text{ such that } \ell' : F' \in T_0 \\ \langle \alpha, F' \rangle & \text{else if there is an } \alpha \text{ such that } \alpha \in T_0 \end{cases}
\end{aligned}$$

Figure 4.13: Definition of $\text{hopper}_H(C)$, where the call-site metadata is $H = T_0; \bar{T}; \bar{\ell} : \bar{F}$.

The hopper function. The function hopper_H (Figure 4.13) is used by the TL abstract machine in search mode, each time the search walks over a call frame (rule T-UNW-APP in Figure 4.9). At that moment the clue describes the handler’s provenance from the callee’s perspective; the hopper re-expresses it from the caller’s perspective. It can do so because the call-site metadata H records how the caller instantiated each of the callee’s binders. The definition of the hopper is a case analysis on the first component of the clue, a TL index. The second component, an effect name, is kept unchanged by the hopper.

If the index is a label index $\hat{i}$, the handler reached the callee as its i -th label parameter. How the caller instantiated that parameter is recorded in the third component of H : its i -th entry $\ell':F'$ gives the caller-side index ℓ' . The new clue carries ℓ' in place of $\hat{i}$; it refers to the same handler but locates it relative to the caller's frame.

If the index is a capability index $\overset{\circ}{i}$, the handler was hidden behind the callee's i -th capability variable, so the hopper inspects the capability $\bar{T}^{(i)}$ instantiating it. Two outcomes are possible: either $\bar{T}^{(i)}$ contains a label index whose effect name matches the clue, and that label will be chosen; or the handler is hidden one level deeper, behind the capability index in $\bar{T}^{(i)}$, and the new clue follows that capability index.

If the index is ∞ , the handler was lexically captured by the callee. The first component T_0 of H records exactly the variables the callee captures, so the hopper searches T_0 just as it searches $\bar{T}^{(i)}$ in the previous case.

Notice that the last two cases rely on the non-ambiguity and non-frivolity conditions required of an abstraction: together they guarantee that, at run time, the capture set of any function that is called contains labels of distinct effect names (a guarantee established by the subject-reduction development in Appendix C.1). So as soon as a label index matching the effect name in the clue is found, it is guaranteed to identify the desired handler, because no other label index with the same effect name can exist in the capability being searched ($\bar{T}^{(i)}$ or T_0) or in the further instantiation of any capability index within it.

The hopper is a partial function: it is undefined when a clue resolves to neither a unique matching label index nor a capability index, which would leave the TL machine stuck. The simulation result of Section 4.4 guarantees that this never happens for a TL program produced from a well-typed SL program. Finally, the restriction that a capability contain at most one capability variable is what keeps the hopper deterministic: when no label matches, there is only one capability index to follow. The restriction can be lifted by letting a clue carry a list of indices, so that the stackwalker can follow several capability variables at once.

4.4 Compiler Correctness

We prove the translation correct by showing that it preserves semantics. Throughout, we use colour to distinguish SL (blue) from TL (orange). The central result is a simulation theorem, from which semantics preservation follows.

$$\begin{aligned}
\text{value } v &::= \dots \mid \{T\} \left[\bar{\alpha}; \bar{\ell}; F \right] (\bar{x}) \Rightarrow t \mid \text{cont } \{\bar{L}_F\} \# L_F^{(x,k) \Rightarrow t} K \\
\text{expression } e &::= \dots \mid v_1 \left[T; \bar{T}; \bar{l} \right] (\bar{v}_2) \mid \{T\} \text{handle } [\ell : F] \Rightarrow t_1 \text{ with } (x, k) \Rightarrow t_2 \\
&\quad \mid \text{resume } v_1 [T] (v_2) \\
\text{label } l &::= \ell \mid L_F \\
\text{frame } A &::= \dots \mid \text{let } x = \left(\{\bar{L}_F\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1
\end{aligned}$$

Figure 4.14: Selected syntax of SL^+ . The difference between SL^+ and SL is highlighted.

Theorem 4.1 (Simulation). *Given a well-typed SL^+ configuration M (where SL^+ , defined in this section, enriches SL with runtime constructs) and a TL configuration M such that $M \sim M$, if $M \longrightarrow M'$, then there exists a TL configuration M' such that $M \longrightarrow^* M'$ and $M' \sim M'$.*

Corollary 4.2 (Semantics Preservation). *If $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash t : \tau \rightsquigarrow t$ and $\langle \epsilon \parallel t \rangle \longrightarrow^* \langle \epsilon \parallel v \rangle$, then there exists v such that $\epsilon \mid \epsilon \mid \epsilon \mid \epsilon \vdash v : \tau \rightsquigarrow v$ and $\langle \epsilon \parallel t \rangle \longrightarrow^* \langle \epsilon \parallel v \rangle$.*

Corollary 4.2 states that if a closed, well-typed SL program terminates with a value, then its TL translation also terminates and produces the corresponding value. Its proof follows the standard forward-simulation framework of Leroy [26]: one establishes a condition on initial states, a condition on final states (both straightforward), and the simulation theorem itself.

The simulation relation $\sim$ is the technical heart of the proof. The remainder of this section defines the enriched source syntax the proof works with and presents selected rules of the relation.

An enriched source syntax. The proof is carried out over an enriched version of SL , which we call SL^+ , in which values and expressions carry additional type-level annotations; the annotations are recoverable from the typing of SL programs and make the relations easier to state. In SL^+ (Figure 4.14), abstractions and **handle** expressions carry the capture set $\{T\}$ that in SL exists only in types. Continuation values also carry a capture set; because continuations are created at run time, it takes the form of a list of run-time labels. Applications and **resume** expressions are enriched with the capture set of the function and continuation, respectively, and a run-time label L carries its effect name, as in L_F (omitted when not relevant).

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

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

Promises relate TL indices to the SL^+ labels and capabilities that come from an abstraction's parameters or capture set. A promise Π has three components: $\tilde{\Sigma}$, the label promise, relates TL label indices to SL^+ label variables; $\tilde{\Delta}$, the capability promise, relates TL capability indices to SL^+ capability variables; and $\tilde{\Theta}$, the capture promise, is either a single SL^+ capability variable or a list of SL^+ labels with their effect names. While $\tilde{\Sigma}$ and $\tilde{\Delta}$ may contain only SL^+ variables, $\tilde{\Theta}$ may contain both variables and run-time labels. We write $\emptyset$ for a promise whose three components are all empty.

Evidences relate TL clues to SL^+ run-time labels: having an evidence $\textcolor{brown}{C} \mapsto \textcolor{blue}{L}$ in the relational context means that the TL abstract machine in search mode carrying clue $\textcolor{brown}{C}$ will find the handler corresponding to the SL^+ label $\textcolor{blue}{L}$. Since evidences relate run-time labels, they are a more dynamic notion than promises: programs are initially related through promises, and as they evaluate, call frames and handler frames contribute evidence.

Label and capability relations. $\boxed{l \overset{\textcolor{blue}{l}}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Sigma}} \textcolor{brown}{C}} \quad \boxed{T \overset{\textcolor{blue}{T}}{\sim}_{\Xi|\Pi} \textcolor{brown}{T}} \quad \boxed{T_0; \bar{T}; \bar{l} \overset{\textcolor{blue}{H}}{\sim}_{\Xi|\Pi} H}$

The definitions of the label and capability relations appear in Appendix C.2. Informally, an SL^+ label $\textcolor{blue}{l}$ is related to a TL clue $\textcolor{brown}{C}$ if either the promise or the evidence relates them. The relation between capabilities $\textcolor{blue}{T}$ and $\textcolor{brown}{T}$, as well as that between $T_0; \bar{T}; \bar{l}$ and call-site metadata H , are defined structurally; recall that H in TL has a three-tuple structure matching $T_0; \bar{T}; \bar{l}$ in SL.

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

$$\begin{aligned}
\text{promise } (T; \bar{\alpha}; \overline{\ell:F}) &= T; \left\{ \bar{\alpha}^{(i)} \sim \hat{i} \mid i \right\}; \left\{ \overline{\ell:F}^{(i)} \sim \hat{i} \mid i \right\} \\
\text{evidence } (\overline{L_0 F_0}; \overline{L_1 F_1}; \overline{L_2 F_2}) &= \left\{ \left\langle \infty, \overline{F_0}^{(i)} \right\rangle \mapsto \overline{L_0}^{(i)} \mid i \right\} \cup \left\{ \left\langle \hat{i}, \overline{F_1}^{(i,j)} \right\rangle \mapsto \overline{L_1}^{(i,j)} \mid i, j \right\} \\
&\quad \cup \left\{ \left\langle \hat{i}, \overline{F_2}^{(i)} \right\rangle \mapsto \overline{L_2}^{(i)} \mid i \right\}
\end{aligned}$$

$$\begin{array}{c}
\text{R-V-FUN} \quad \frac{\begin{array}{c} t \quad \text{term} \quad t \\ \emptyset | \text{promise}(T; \bar{\alpha}; \bar{\ell}F) \end{array}}{\{T\} [\bar{\alpha}; \bar{\ell}:F] (\bar{x}) \Rightarrow t \stackrel{\text{val}}{\sim} (\bar{x}) \Rightarrow t} \\
\\
\text{R-E-APP} \quad \frac{\begin{array}{c} v_1 \stackrel{\text{val}}{\sim} v_1 \quad \bar{v}_2 \stackrel{\text{val}}{\sim} \bar{v}_2 \quad T_0; \bar{T}; \bar{l} \stackrel{\sim}{\sim} H \\ \Xi | \Pi \end{array}}{v_1 [T_0; \bar{T}; \bar{l}] (\bar{v}_2) \stackrel{\text{expr}}{\sim}_{\Xi | \Pi} (v_1(\bar{v}_2))^H} \\
\\
\text{R-V-CONT} \quad \frac{\begin{array}{c} t \stackrel{\text{term}}{\sim}_{\Xi | \emptyset} t \quad K \quad \text{cont} \quad K \quad \Xi = \text{evidence}(\bar{L}'_{F'}; \epsilon; \epsilon) \\ \Xi \cup \{ \langle \bar{\theta}, F \rangle \mapsto L \} \end{array}}{\text{cont } \{ \bar{L}'_{F'} \} \# L_F^{(x,k) \Rightarrow t} K \stackrel{\text{val}}{\sim} \text{cont } \#^{(x,k) \Rightarrow t} K} \\
\\
\text{R-E-RAISE} \quad \frac{\begin{array}{c} l \stackrel{l}{\sim}_{\Xi | \bar{\theta}; \bar{\Sigma}} C \quad v \stackrel{\text{val}}{\sim} v \end{array}}{\text{raise } l(v) \stackrel{\text{expr}}{\sim}_{\Xi | \bar{\theta}; \bar{\Delta}; \bar{\Sigma}} \text{raise } C(v)}
\end{array}$$

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

R-E-APP states that an SL^+ application is related to a TL application if the functions v_1 and v_1 are related, the arguments $\bar{v}_2$ and $\bar{v}_2$ are related, and also $T_0; \bar{T}; \bar{l}$ and the TL call-site metadata H are related. The last premise effectively guarantees that all the capabilities this function application needs are either promised to be related by Π or already related by Ξ .

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

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

These relations are also indexed by a context relating free term variables, which we omit for conciseness.

Evaluation-context relation. $\boxed{E \stackrel{\text{ctx}}{\sim}_{\Xi} E}$

$$\begin{array}{c}
\frac{\forall C \in \text{dom}(\Xi'). \Xi(\text{hopper}_H(C)) = \Xi'(C)}{\text{truthful}_{\Xi|H}(\Xi')} \\
\\
\text{R-EC-APP} \quad \frac{E \stackrel{\text{ctx}}{\sim}_{\Xi} E \quad t \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t \quad \text{truthful}_{\Xi|H}(\Xi')}{E \cdot (\text{let } x = \square \text{ in } t) \stackrel{\text{ctx}}{\sim}_{\Xi'} E \cdot (\text{let } x = \square^H \text{ in } t)} \\
\\
\text{R-EC-HANDLER} \quad \frac{E \stackrel{\text{ctx}}{\sim}_{\Xi} E \quad t_2 \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t_2 \quad t_1 \stackrel{\text{term}}{\sim}_{\Xi'|\emptyset} t_1 \quad \text{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi') \quad \Xi' = \text{evidence}(\overline{L'_{F'}}; \epsilon; \epsilon)}{E \cdot (\text{let } x = \{ \overline{L'_{F'}} \} \# L_F^{(y,k) \Rightarrow t_1} \square \text{ in } t_2) \stackrel{\text{ctx}}{\sim}_{\Xi' \cup \{ \langle \bar{0}, F \rangle \mapsto L \}} E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_1} \square)^T \text{ in } t_2)}
\end{array}$$

The evaluation-context relation uses a judgement $\text{truthful}_{\Xi|H}(\Xi')$, which means that all the evidence mappings in Ξ' can be found in Ξ with an extra indirection through hopper_H ; this is precisely where the correctness of a hopper step is discharged. In the expression and term relations, the evidence Ξ can be viewed as input. By contrast, in the evaluation-context relation, the evidence should be viewed as output, which serves as the input for relating expressions and terms plugged into the evaluation context.

R-EC-APP states that if an evaluation context E in SL^+ is related to an evaluation context E in TL and yields an evidence Ξ , then the contexts extended with the frames of a function call (a let frame in SL^+ , a call frame in TL) are related and yield an evidence Ξ' , as long as Ξ' is truthful with respect to Ξ and the call-site metadata H on the frame.

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

Simulation proof. Theorem 4.1 is proved by case analysis on the SL^+ transition relation $M \longrightarrow M'$. The full definition of the simulation relation and the proof can be found in Appendices C.2 and C.3.

4.5 Implementation

We implemented the design in the LEXA compiler [55]. LEXA is a functional language with lexical effect handlers whose compiler uses segmented stacks to capture and restore continuations efficiently [33] (Chapter 2) and represents labels as stack addresses, which makes handler search a constant-time operation but incurs a small overhead on mainline paths. We refer to that original strategy as *Direct LEXA* and to the new strategy as *Zero LEXA*. The extended language lets the programmer declare effects as *exceptional*: exceptional effects, expected to be raised infrequently, are compiled with Zero LEXA, while other effects use Direct LEXA, and the two can coexist in one program. The LEXA compiler is written in OCaml and compiles to C, which LLVM then optimizes; since Zero LEXA differs from Direct LEXA only in how the runtime locates handlers, it reuses the same pipeline.

4.5.1 Hopper Table

At the core of the formal model are the hopper_H functions, which guide handler search frame by frame. Evaluating a hopper at run time might seem expensive, but each hopper instance depends only on compile-time typing information, so we precompute each one into a static lookup table mapping a clue to the next clue. Since each hopper instance is uniquely associated with a call site, we build a global table mapping the *return address* of each call site to its hopper; the table also stores frame sizes, needed for walking the stack. It lives in the program’s data section, completely hidden from the mainline paths.

Figure 4.15 shows how the hopper for the call site at line 10 of Figure 4.3 is compiled. The call-site information records `fun1`’s capture set (ℓ_1) and the instantiations of its capability variable α_1 (with (ℓ_0, α_0)) and label variable ℓ_2 (with ℓ_0); the translation expresses each as a TL index. The resulting hopper is a table whose rows are possible effect names and whose columns are possible clue indices (∞ for the capture set, plus one column per parameter); each cell gives the output clue, with NA marking an impossible input.

This table is then flattened into bits. Each entry of the global hopper table consists of the call site’s return address, its frame size, and the bit-encoded hopper. Assuming that a program defines at most 16 effect signatures and that at most 16 label or capability variables are in scope at any point, we use four bits each for the effect name and the index, plus a two-bit tag distinguishing the three index forms ($\hat{i}$, $\overset{\circ}{i}$, ∞). Each entry also carries a one-bit flag indicating whether the caller is invoking a tail-resumptive handler in

	capture set	capability inst.	label inst.		∞	$\hat{0}$	$\hat{0}$
in SL	ℓ_1	(ℓ_0, α_0)	ℓ_0	F_0	NA	$\langle \hat{1}, F_0 \rangle$	$\langle \hat{1}, F_0 \rangle$
in TL	$\hat{0}:F_1$	$(\hat{1}:F_0, \hat{0})$	$\hat{1}:F_0$	F_1	$\langle \hat{0}, F_1 \rangle$	$\langle \hat{0}, F_1 \rangle$	NA

(a) Call-site information for line 10.

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

(b) Tabular representation of the hopper.

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

(c) Binary representation of the hopper.

Figure 4.15: Compiling a hopper instance for the call site at line 10 of Figure 4.3: (a) the call-site information, (b) the resulting hopper as a table, and (c) its binary representation. For illustration, the tables assume that ℓ_0 and ℓ_1 carry distinct effect names F_0 and F_1 ; in Figure 4.3 itself both have the single effect name F .

place (Section 4.5.2). Across our benchmarks, the hopper table takes up only a single-digit percentage of the binary size.

4.5.2 Stackwalker, Multishot Resumptions, and Tail-Resumptive Handlers

The stackwalker is a function that takes an initial clue from the **raise** site. On arriving at a frame, it uses the frame’s return address to fetch a new clue using the global hopper table, then either locates the handler in the current frame or jumps to the next frame, repeating until the handler is found.

Because Zero LEXA needs no run-time representation of labels, multiple copies of the same resumption do not interfere, so the strategy supports multishot resumptions without extra effort, in contrast to Direct LEXA, whose stack addresses go stale under stack copying (the very problem Chapter 3 solves differently). We do not regard this as a key advantage of Zero LEXA, however, since applications that need multishot resumptions are usually effect-heavy and do not benefit from the trade-off Zero LEXA is designed for.

Tail-resumptive handlers require care. A tail-resumptive handler invokes its resumption as its final action, so it can be run in place like an ordinary function call, avoiding continuation capture. But if such an in-place handler then raises an effect, that effect must be handled in the context where the handler was *installed*, which may be higher up the call

chain; a stackwalker starting from the current frame would find the wrong handler. We address this in two places. When the program invokes a tail-resumptive handler in place, the handler’s clue is pushed onto the stack before the call. When the stackwalker reaches a frame, it checks a special flag in the hopper entry; if the caller is invoking a tail-resumptive handler, it loads the saved clue and walks to the frame where that handler was installed before resuming the walk with its original clue. Since tail-resumptive call sites can nest, the stackwalker keeps a stack of clues, pushing and popping as it meets tail-resumptive call sites and install sites.

Separate compilation. Zero LEXA is compatible with separate compilation: each compilation unit produces its own hopper table, and at link time the tables are merged with code addresses updated from relocation information. We leave a full implementation of this to future work.

4.6 Evaluation

We study the performance of the two strategies as implemented in the LEXA compiler. Direct LEXA makes handler search and continuation capture efficient but incurs a small overhead even when effects are not raised, violating the zero-overhead principle. Zero LEXA gives up some efficiency in handler search in exchange for minimal overhead on mainline paths, and we expect it to win for effects that are raised infrequently.

We use two benchmark sets. The first is a community-maintained suite [15] of effect-heavy programs that stress handler search and continuation capture; on these, LEXA uses Direct LEXA by default and delivers competitive results, and integrating the zero-overhead strategy does not degrade that code. No existing suite stresses the efficiency of mainline code, so we curated a second set of *effect-infrequent* benchmarks that are similar in size and complexity but raise effects infrequently; most use exceptions to signal invalid input or other conditions not expected to occur. Table 4.1 reports both sets. For the first set, the **Zero** column shows performance when the zero-overhead strategy is *forced*; for the second set, it shows LEXA’s natural choice for exceptional effects.

4.6.1 Effect-Heavy Benchmarks

The first set comes from the community suite [15]. As expected, LEXA’s performance degrades significantly when forced to use Zero LEXA, because stackwalking becomes a

Table 4.1: Two benchmark sets: an existing suite of effect-heavy programs (top) and a new suite of effect-infrequent programs (bottom). Running times are in milliseconds. “Loss/Gain” is Zero LEXA relative to Direct LEXA; the last two columns give the hopper table as a percentage of binary size and stackwalking as a percentage of total running time. “−Inf” marks a benchmark that Direct LEXA optimizes to a constant time of 0. All experiments were conducted on a workstation with an AMD Ryzen 7 CPU (4.5 GHz).

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

major overhead when effects are raised frequently. **Countdown** and **Iterator** are optimized to constants by LLVM under Direct LEXA, but the stackwalking logic prevents the same inlining under Zero LEXA. **Parsing Dollars** has nested handlers, so the stackwalker must walk over many frames, whereas finding a handler is constant-time under Direct LEXA. **Handler Sieve** uses a chain of tail-resumptive handlers, whose in-place invocation under Zero LEXA requires the extra clue-stack machinery of Section 4.5.2 and incurs a large overhead.

4.6.2 Effect-Infrequent Benchmarks

The second set consists of programs whose inputs produce no effects, or trigger them only infrequently. **Catalan**, **Bézout**, **Golomb**, **Hofstadter Q**, and **Karatsuba** use exceptions to indicate invalid input (for example, a non-positive modulus) that does not arise for the inputs we use. Even so, under Direct LEXA the exception handlers need run-time representations, and passing them around through these recursive programs is a major overhead; Zero LEXA avoids it and is faster. **Ackermann** and **Palindrome Partition** show no clear advantage for Zero LEXA: the programs are simple enough that the CPU absorbs the cost of the extra Direct LEXA argument through out-of-order execution. **Lattice Path** contains a function that already takes six arguments, so Direct LEXA’s additional handler argument introduces noticeable overhead that Zero LEXA avoids.

Two Threads runs two lightweight threads that cooperatively multitask on one OS thread; each holds a time-slice counter and raises a **Yield** effect when it reaches zero. The time slice, given as input, controls how often effects are raised. When the slice is large, the threads yield infrequently, the cost of propagating the **Yield** handler dominates, and Zero LEXA wins (as in Table 4.1); when the slice is small, stackwalking dominates and Direct LEXA wins. Figure 4.16 plots the running times against the time slice: as the slice increases, the advantage shifts from Direct LEXA to Zero LEXA.

4.6.3 Hopper Size and Stackwalking Time

The last two columns of Table 4.1 report the space and time costs of Zero LEXA. The increase in binary size from the hopper table is modest, a single-digit percentage across all benchmarks. Stackwalking time varies widely: it dominates most of the effect-heavy benchmarks but is negligible in the effect-infrequent ones. These results confirm that the zero-overhead strategy is best suited to scenarios where effects are raised infrequently or not at all; we expect future engineering to reduce the cost of stackwalking further.

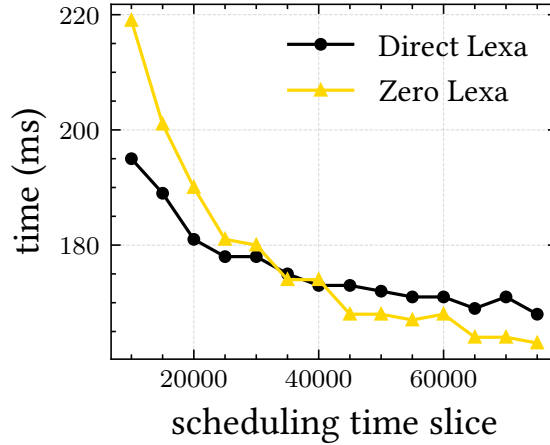


Figure 4.16: Running time of the **Two Threads** benchmark as a function of the time slice. As the slice grows, the **Yield** effect is raised less frequently, shifting the advantage from Direct LEXA to Zero LEXA.

4.7 Related Work

Almost all languages with exceptions use dynamic scoping for their handlers. CLU is among the earliest [1, 29]: its compiler generates exception tables consulted at run time to find the closest dynamically enclosing handler, an approach later used in Java and most C++ implementations. Dynamic scoping is flexible but hazardous. Zhang et al. [69] observe that dynamically scoped exception handlers can catch exceptions by accident and address this in Genus [67] with *tunneled exceptions* (a form of lexical scoping) implemented by a translation to Java exceptions that generates fresh identifiers for each handler instance, and so does not satisfy the zero-overhead principle. Zhang et al. [68, 70] argue that the unintended interception of exceptions is a symptom of a deeper modularity problem, understood in analogy to a loss of *parametricity* [59], and prove that lexical scoping restores local-reasoning principles; in our SL, effect polymorphism appears as parameterization over capability variables. Biernacki et al. [4] coined the term *lexically scoped effect handlers* and showed that generativity is necessary when effect operations can be polymorphic. Our work is the first to implement lexical handlers without generating any reified representation of handlers at all.

Effekt supports lexical handlers with lightweight effect polymorphism via second-class values [7]. A version of its compiler translates to an intermediate language where handlers are passed explicitly, then performs *lift inference* to generate subregion evidence determin-

ing how many handlers to skip over when an effect is raised [38, 45]; reifying this evidence as a sequence of function applications violates the zero-overhead principle. A more recent version of Effekt instead builds on LEXA’s approach of compiling to stack switching [51, 33]. LEXA [33] (Chapter 2) achieves constant-time handler search and resumption capture by compiling to stack switching, at the cost of reifying handler instances as stack addresses; the present work extends LEXA with a different trade-off aimed at low-frequency effects. Koka’s named handlers [65] are a form of lexical handlers that reify handler instances as evidence at run time, and so also do not satisfy the zero-overhead principle. An ongoing effort to add effect handlers to WebAssembly, WasmFX [40], focuses on dynamic scoping but also considers named handlers, without studying their implementation.

4.8 Conclusion

Lexically scoped handlers address a long-standing modularity problem with dynamic scoping, but existing implementations do not meet the zero-overhead principle for exceptional effects, a principle otherwise satisfied by most modern implementations of dynamically scoped exception handling. This chapter introduced a compilation strategy that restores the principle through a type-directed translation that emits information tracking the lexical provenance of handlers. That information guides a stackwalker, steered by per-call-site hops, to the right handler for a raised effect. The payoff is reduced overhead on main-line paths, achieving parity with the performance profile of dynamically scoped exception handlers. The approach complements the strategies of Chapters 2 and 3, letting the compilation strategy for each declared effect be matched to its expected invocation frequency. By identifying and removing a key barrier to adoption, we hope this work encourages language designers and implementers to embrace lexically scoped handlers for their modularity benefits.

With Zero LEXA, the implementation techniques of this thesis span the full range of effect-usage patterns, from the effect-heavy workloads served by Chapters 2 and 3 to the effect-infrequent workloads served here. Chapter 5 draws the three techniques together.

Chapter 5

Conclusion

Lexically scoped effect handlers recover the modular reasoning that dynamic scoping compromises: a handler is a lexically scoped capability, and an effect can be raised only to a handler whose capability is in scope, so no handler can accidentally intercept an effect not meant for it (Chapter 1). However, the inefficiency of implementations prior to the work in this thesis could hinder its adoption in practice. The position of this thesis is that the very property that makes lexical scoping safe, namely its static character, is also what makes it efficient to implement. At a raise site, the handler that will receive the effect is determined by lexical provenance, and a compiler can exploit that fact instead of searching for the handler at run time.

The thesis developed this idea into three implementation techniques, each aimed at a different point in the design space. Direct LEXA (Chapter 2) gives each handler a run-time identity, the stack address at which it is installed, and passes that identity to the sites that raise effects to it, so a raise jumps directly to its handler and the resumption is captured directly as a span of stack. Zone LEXA (Chapter 3) removes the central limitation of the address-based strategy, its incompatibility with multishot resumptions, by *virtualizing* handler identities: a software memory-management unit gives each copy of a resumption its own virtual view of the handlers within it, fully supporting multishot resumptions and even allowing copies of the same resumption to run in parallel. Zero LEXA (Chapter 4) targets a different point in the design space, where the dominant concern is a small but ever-present cost: even executions that never raise an effect pay for passing the run-time handler identity down the call chain. Zero LEXA eliminates that cost by not giving handlers any run-time identity, instead relying on stackwalking to find handlers when effects are raised. The lexical provenance of handlers is recorded as static metadata on the call stack, so the overall behaviour still preserves the semantics of lexical scoping. Each technique is made

precise by a formal translation proven semantics-preserving, and each is realized in the LEXA compiler with measurable performance gains.

These techniques are complementary rather than competing. Direct and Zone LEXA serve effect-heavy workloads, where effects are raised often and the priority is fast handling; Zero LEXA serves effect-infrequent workloads, where the priority is an undisturbed mainline. The choice of strategy is declared per effect, so a single program can mix them, matching each declared effect to its expected usage. Together the three techniques cover the spectrum of effect-usage patterns, and they share a single guiding principle: locate handlers using static, lexical information rather than a dynamic run-time search.

The overarching message of this thesis is a simple one. Adopting lexically scoped effect handlers, for the modularity and abstraction safety they provide, need not mean giving up performance. With the implementation techniques presented here, lexical effect handlers are both safe and fast, and we hope this makes them a natural choice for future programming languages.

References

- [1] Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. Aspects of implementing CLU. In *Proc. of the 1978 Annual Conference (ACM '78)*, 1978.
- [2] Andrej Bauer and Matija Pretnar. 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. Handle with care: Relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018.
- [4] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2020.
- [5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9), 1988.
- [6] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. Effects, capabilities, and boxes: From scope-based reasoning to type-based reasoning and back. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022.
- [7] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.
- [8] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming*, 30, 2020.

- [9] Will Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *ACM Conf. on LISP and Functional Programming (LFP)*, 1988.
- [10] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, 1999.
- [11] Olivier Danvy and Andrzej Filinski. Abstracting control. In *ACM Conf. on LISP and Functional Programming (LFP)*, 1990.
- [12] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5), 1972.
- [13] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021.
- [14] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6), 2007.
- [15] Effect Handlers Community. Effect handlers benchmarks suite. <https://github.com/effect-handlers/effect-handlers-bench>. Accessed: 2024-04-01.
- [16] Kavon Farvardin and John Reppy. From folklore to fact: Comparing implementations of stacks and continuations. In *PLDI*, 2020.
- [17] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.
- [18] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *ICFP*, 2007.
- [19] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.
- [20] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.
- [21] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.

- [22] Daniel Hillerström. multicont. <https://opam.ocaml.org/packages/multicont>. Accessed: 2024-07-01.
- [23] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, 2013.
- [24] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *POPL*, 2017.
- [25] Daan Leijen and KC Sivaramakrishnan. libmprompt. <https://github.com/koka-lang/libmprompt>, 2021.
- [26] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4), 2009.
- [27] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *POPL*, 2017.
- [28] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8), 1977.
- [29] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6), 1979.
- [30] Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. In *POPL*, 2006.
- [31] Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. Zero-overhead lexical effect handlers. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2), 2025.
- [32] Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. Zero-overhead lexical effect handlers (extended version). Technical Report CS-2025-04, School of Computer Science, University of Waterloo, 2025.
- [33] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. Lexical effect handlers, directly. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2), 2024.
- [34] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. Lexical effect handlers, directly (extended version). Technical Report CS-2024-04, School of Computer Science, University of Waterloo, 2024.
- [35] Cong Ma, Jonghyun Jung, and Yizhou Zhang. Virtualizing continuations. *Proceedings of the ACM on Programming Languages*, 10(PLDI):211:1–211:23, June 2026.

- [36] Cong Ma, Jonghyun Jung, and Yizhou Zhang. Virtualizing continuations (extended version). Technical Report CS-2026-02, School of Computer Science, University of Waterloo, 2026.
- [37] Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. Multiple resumptions and local mutable state, directly. *Proceedings of the ACM on Programming Languages*, 9(ICFP), 2025.
- [38] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. From capabilities to regions: Enabling efficient compilation of lexical effect handlers. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), 2023.
- [39] Cao Nguyen Pham and Martin Odersky. Stack-copying delimited continuations for Scala Native. In *Proc. of the 19th ACM Int’l Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)*, 2024.
- [40] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing WebAssembly with effect handlers. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), 2023.
- [41] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
- [42] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.
- [43] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.
- [44] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *ACM Conf. on LISP and Functional Programming*, 1992.
- [45] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. A typed continuation-passing translation for lexical effect handlers. In *PLDI*, 2022.
- [46] Mark Shannon. “Zero cost” exception handling. Python issue, <https://bugs.python.org/issue40222>, 2020.

- [47] KC Sivaramakrishnan. Pearls of algebraic effects and handlers. <https://kcsrk.info/ocaml/multicore/effects/2015/05/27/more-effects>, 2015. Blog post.
- [48] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *PLDI*, 2021.
- [49] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [50] Bjarne Stroustrup. Foundations of C++. In *European Symp. on Programming (ESOP)*, 2012.
- [51] The Effekt Team. A brief history of Effekt for fellow researchers. <https://effekt-lang.org/evolution>. Accessed: 2025-07-01.
- [52] The Go Team. Go 1.3 release notes: Stacks. <https://go.dev/doc/go1.3#stacks>, 2014. Accessed: 2025-11-13.
- [53] The Go Team. runtime: new goroutines can spend excessive time in morestack. GitHub issue golang/go#18138, <https://github.com/golang/go/issues/18138>, 2016.
- [54] The Go Team. runtime: better adaptive stack sizing. GitHub issue golang/go#77893, <https://github.com/golang/go/issues/77893>, 2026.
- [55] The Lexa Project. The Lexa programming language. <https://github.com/lexa-lang/lexa>. Accessed: 2026-07-31.
- [56] thunderseethe. Zig is getting so close to stumbling into lexical effect handlers. <https://bsky.app/profile/thunderseethe.dev/post/3ltwortlu3223>, July 2025. Bluesky post.
- [57] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
- [58] Orpheas van Rooij and Robbert Krebbers. Affect: An affine type and effect system. *Proceedings of the ACM on Programming Languages*, 9(POPL), January 2025.
- [59] Philip Wadler. Theorems for free! In *Int’l Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1989.
- [60] Luke Wagner. Should producers/consumers assume throwing is “rare”? WebAssembly/exception-handling issue #19, <https://github.com/WebAssembly/exception-handling/issues/19>, 2017.

- [61] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Int'l Workshop on Types in Compilation (TIC)*, 2000.
- [62] Leo White. Locality and effect reflection. Unpublished draft, 2026.
- [63] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *ACM SIGPLAN Symp. on Haskell*, 2014.
- [64] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020.
- [65] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. First-class names for effect handlers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2), 2022.
- [66] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to C. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 2021.
- [67] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *PLDI*, 2015.
- [68] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.
- [69] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame for safe tunneled exceptions. In *PLDI*, 2016.
- [70] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. Handling bidirectional control flow. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

APPENDICES

Appendix A

Simulation Details for Direct LEXA

This appendix gives the full development of the simulation argument summarized in Section 2.5: the simulation relation between LEXA and SALT configurations, its auxiliary relations, and the case analysis proving Theorem 2.2. The material reproduces the corresponding appendix of the technical report [34].

To set up our simulation proof, we relate the configuration of a LEXA program to that of a SALT program.

Relating configurations: The discussion of this relation can be found in Section 2.5.

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

Relating values: $v \stackrel{\text{value}}{\underset{\Xi}{\sim}} v'$. The rules are straightforward. Labels in LEXA are related to memory locations in SALT via the mapping Ξ , which is the identity except for the handler labels of tail-resumptive and abortive handlers.

$$\begin{array}{cccc} i \stackrel{\text{value}}{\underset{\Xi}{\sim}} i & \mathbf{ns} \stackrel{\text{value}}{\underset{\Xi}{\sim}} \mathbf{ns} & P \stackrel{\text{value}}{\underset{\Xi}{\sim}} P & L \stackrel{\text{value}}{\underset{\Xi}{\sim}} \Xi(L) \end{array}$$

Relating a local environment to a stack segment: $E \stackrel{\text{local env}}{\sim} s$. A local environment in LEXA is related to a stack segment in SALT.

$$\frac{\forall i \in \{1, \dots, n\}. v_i \stackrel{\text{value}}{\sim} v_i}{[x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \stackrel{\text{local env}}{\sim} \text{nil} :: v_1 :: \dots :: v_n}$$

Relating a term under a local environment to an instruction sequence: $(E, t) \stackrel{\text{ins}}{\sim} I$. A LEXA term is related to the sequence of SALT instructions produced by the translation of the term, appended with instructions to free the stack segment allocated for the frame.

$$\frac{k = |E| + |t|}{(E, t) \stackrel{\text{ins}}{\sim} \llbracket t \rrbracket_{\text{dom}(E)} ; \text{sfree } k ; \text{return}}$$

Relating an activation frame to a stack segment: $(E, \text{let } x = \square \text{ in } t) \stackrel{\text{frame}}{\sim} s$. Using the two preceding relations, we can relate an activation frame in LEXA and a stack segment in SALT. The stack segment consists of the stack for the local environment and a return address. The return address points to the instruction sequence I to which the continuation t of the current frame is related.

$$\frac{(E, t) \stackrel{\text{ins}}{\sim} I \quad M(P) = \iota_0 :: \dots :: \iota_{i-1} :: I \quad E \stackrel{\text{local env}}{\sim} s}{(E, \text{let } x = \square \text{ in } t) \stackrel{\text{frame}}{\sim} s :: P^i}$$

Relating a sequence of activation frames to a stack segment: $\mathbb{F} \stackrel{\text{frames}}{\sim} s$. The metavariable $\mathbb{F}$ ranges over a subset of LEXA evaluation contexts that are composed of activation frames only:

$$\mathbb{F} ::= \square \mid \mathbb{F} \cdot (E, \text{let } x = \square \text{ in } t)$$

A sequence of activation frames in LEXA simply corresponds to the concatenation of their corresponding stack segments in SALT. Here, we use the associative operator $@$ to denote the concatenation of two stack segments. When the right operand begins with the stack-base marker **nil**, the marker is dropped by the concatenation, so an assembled stack

contains a single **nil** at its base.

$$\frac{}{\square \stackrel{\text{frames}}{\sim} \text{nil}} \quad \frac{\mathbb{F} \stackrel{\text{frames}}{\sim} s \quad (E, \text{let } x = \square \text{ in } t) \stackrel{\text{frame}}{\sim} s'}{\mathbb{F} \cdot (E, \text{let } x = \square \text{ in } t) \stackrel{\text{frames}}{\sim} s @ s'}$$

Relating a sequence of handler-led contexts to stacks:

$$\downarrow \mathbb{K} \stackrel{\text{handlers}}{\sim} \left(\begin{array}{ccccc} \downarrow & \downarrow & \uparrow & \uparrow & \uparrow \\ L_{\text{in}}, s_{\text{in}}, & H_{\text{stacks}}, & L_{\text{out}}, & s_{\text{out}} \end{array} \right)$$

The metavariable $\mathbb{K}$ ranges over a subset of LEXA evaluation contexts that are composed of a sequence of activation frames but also have a handler frame in their outermost layer:

$$\mathbb{K} ::= \left(\#L_A^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F}$$

We call such evaluation contexts *handler-led*. We use $\mathbb{K}$ to mean a sequence of handler-led contexts, with $\square$ overloaded to mean the innermost hole in a handler-led context:

$$\mathbb{K} ::= \square \mid \mathbb{K} \cdot \mathbb{K}$$

The relation between a sequence of handler-led contexts and a set of stacks is defined inductively. It is complicated for two reasons. First, 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. Second, for a general handler, its header frame is on a new stack, with the exchanger pointing to the top of the previous stack. So the stacks need to be constructed one by one, taking the previously constructed stack as input. For this purpose, the relation takes in L_{in} , the base 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 output H_{stacks} is the set of stacks that have completed construction. Below we present the inference rules one by one.

The base case is when the sequence of handler-led contexts is just the innermost hole. In this case, the input and the output are the same stack L , and no stack is added to the heap, as there may be further construction to be done on the stack L .

$$\frac{}{\square \stackrel{\text{handlers}}{\sim} (L, s, \{\}, L, s)}$$

The next three cases are the inductive cases, one for each type of handler. For a general handler, the input stack L_{in} is added to the set of constructed stacks, and the construction of a new stack L is started. The exchanger of this new stack points to $L_{\text{in}}^{|s_{\text{in}}|}$, the top of the previous stack. L is used as the input stack for relating the subsequence of handler-led contexts $\mathbb{K}$ to H_{stacks} , which may (or may not) further grow the stack based at L .

$$\frac{\begin{array}{c} \mathbb{F} \stackrel{\text{frames}}{\sim}_{\Xi} s \quad s' = \text{nil} :: P_{\text{op}} :: L_{\text{env}} :: 0 :: L_{\text{in}}^{|s_{\text{in}}|} :: P_{\text{handle}}^{10} @ s \\ \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L, s', H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}}) \quad \Xi(L) = L \end{array}}{\left(\#L_{\text{general}}^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F} \cdot \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L_{\text{in}}, s_{\text{in}}, H_{\text{stacks}} \uplus \{L_{\text{in}} \mapsto s_{\text{in}}\}, L_{\text{out}}, s_{\text{out}})}$$

For a tail-resumptive or abortive handler, the input stack is further grown to include the header frame of the handler. The only difference between the following two rules is the handler annotation stored in the header frame.

$$\frac{\begin{array}{c} \mathbb{F} \stackrel{\text{frames}}{\sim}_{\Xi} s \quad s' = s_{\text{in}} :: P_{\text{op}} :: L_{\text{env}} :: 1 :: \text{ns} :: P_{\text{handle_special}}^8 @ s \\ \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L_{\text{in}}, s', H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}}) \quad \Xi(L) = L_{\text{in}}^{|s_{\text{in}}|} \end{array}}{\left(\#L_{\text{tail}}^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F} \cdot \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L_{\text{in}}, s_{\text{in}}, H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}})}$$

$$\frac{\begin{array}{c} \mathbb{F} \stackrel{\text{frames}}{\sim}_{\Xi} s \quad s' = s_{\text{in}} :: P_{\text{op}} :: L_{\text{env}} :: 2 :: \text{ns} :: P_{\text{handle_special}}^8 @ s \\ \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L_{\text{in}}, s', H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}}) \quad \Xi(L) = L_{\text{in}}^{|s_{\text{in}}|} \end{array}}{\left(\#L_{\text{abort}}^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F} \cdot \mathbb{K} \stackrel{\text{handlers}}{\sim}_{\Xi} (L_{\text{in}}, s_{\text{in}}, H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}})}$$

Relating an evaluation context and a local environment to stacks and the stack pointer:

$$(K, E) \stackrel{\text{context}}{\sim}_{\Xi} (H_{\text{stacks}}, \ell_{\text{sp}})$$

This relation captures the notion of the current point of control in the two languages. A LEXA evaluation context K can be viewed as consisting of a (possibly empty) sequence of activation frames $\mathbb{F}$ followed by a (possibly empty) sequence of handler-led contexts $\mathbb{K}$. The relation also relates a local environment E ; its corresponding stack segment s' is the top segment of the current stack. The output stack s_{out} , corresponding to $\mathbb{K}$, is grown to include s' . With the stack pointer pointing to the top of $s_{\text{out}} @ s'$, the construction of the

stack at L_{out} is complete, so the stack is added to the heap. Also notice that we use L_{init} as a distinguished location for the base of the initial stack.

$$\frac{\mathbb{F} \stackrel{\text{frames}}{\sim} s \quad \mathbb{K} \stackrel{\text{handlers}}{\sim} (L_{\text{init}}, s, H_{\text{stacks}}, L_{\text{out}}, s_{\text{out}}) \quad E \stackrel{\text{local env}}{\sim} s' \quad s'' = s_{\text{out}} @ s'}{(\square \cdot \mathbb{F} \cdot \mathbb{K}, E) \stackrel{\text{context}}{\sim} (H_{\text{stacks}} \uplus \{L_{\text{out}} \mapsto s''\}, L_{\text{out}}^{|s''|})}$$

Relating a heap value to a heap: $(L, V) \stackrel{\text{heap val}}{\sim} H$. A heap value in LEXA is either a tuple or a resumption. The relation for tuples is straightforward.

$$\frac{\forall i \in \{1, \dots, n\}. v_i \stackrel{\text{value}}{\sim} v_i}{(L, \langle v_1, \dots, v_n \rangle) \stackrel{\text{heap val}}{\sim} \{L \mapsto \langle v_1, \dots, v_n \rangle\}}$$

The relation for resumptions is more involved. Notice that with the optimization for abortive and tail-resumptive handlers, resumptions in LEXA are reified only for general handlers. So a resumption in a LEXA heap must be of the form

$$\text{cont} \left(\#L_{\text{general}}^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F} \cdot \mathbb{K}'$$

The relation for handler-led contexts is used to relate the evaluation context $\mathbb{K}$ captured by the resumption to a set H_{stacks} of stacks. The output stack L' also serves as the input stack; the exchanger of the resumption stack L should point to the top of the output stack. Finally, L_{rsp} points to a single-element tuple, which contains the location L^4 of the header frame of the L stack.

$$\frac{\mathbb{K} = \left(\#L_{\text{general}}^{P_{\text{op}}, L_{\text{env}}} \square \right) \cdot \mathbb{F} \cdot \mathbb{K}' \quad \mathbb{K} \stackrel{\text{handlers}}{\sim} (L', s', H_{\text{stacks}}, L', s') \quad \Xi(L) = L}{(L_{\text{rsp}}, \text{cont } \mathbb{K}) \stackrel{\text{heap val}}{\sim} H_{\text{stacks}} \uplus \{L_{\text{rsp}} \mapsto \langle L^4 \rangle\}}$$

Relating data memories: $H \stackrel{\text{data}}{\sim} H$.

$$\frac{\forall i \in \{1, \dots, n\}. (L_i, V_i) \stackrel{\text{heap val}}{\sim} H_i}{\{L_1 \mapsto V_1, \dots, L_n \mapsto V_n\} \stackrel{\text{data}}{\sim} H_1 \uplus \dots \uplus H_n}$$

Relating code memories: $M \stackrel{\text{code}}{\sim} M$. LEXA functions are related to the SALT instruction sequences they are translated to, with the built-in trampoline functions $M_{\text{intrinsic}}$ included.

$$\frac{\forall i \in \{1, \dots, n\}. \llbracket \lambda \overline{x_i}. t_i \rrbracket = I_i}{\{P_1 \mapsto \lambda \overline{x_1}. t_1, \dots, P_n \mapsto \lambda \overline{x_n}. t_n\} \stackrel{\text{code}}{\sim} \{P_1 \mapsto I_1, \dots, P_n \mapsto I_n\} \uplus M_{\text{intrinsic}}}$$

Proof of Theorem 2.2 To prove Theorem 2.2, we need to show that the configuration relation $\sim$ satisfies the three conditions listed in Theorem 2.1, which we restate here for convenience:

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

The initial and final predicates, described informally in Section 2.5, are defined as follows:

- $\text{initial}_{\text{LEXA}}(\text{letrec } \overline{P_i} = \lambda \overline{x_i}. t_i, \langle \{ \overline{P_i} \mapsto \lambda \overline{x_i}. t_i, P_{\text{init}} \mapsto \lambda(). t_{\text{init}} \} \parallel \{ \} \parallel \square \parallel \square \parallel t_{\text{init}} \rangle)$, where t_{init} is defined as $\text{let } x = P_{\text{main}}() \text{ in let } _ = \text{exit } x \text{ in ns end}$
- $\text{initial}_{\text{SALT}}(\overline{P_i} : \overline{I_i}, \langle \{ \overline{P_i} \mapsto \overline{I_i}, P_{\text{init}} \mapsto I_{\text{init}} \} \parallel \{ L_{\text{init}} \mapsto \text{nil} \} \parallel \{ \text{sp} \mapsto L_{\text{init}}, \text{ip} \mapsto P_{\text{init}} \} \rangle)$, where I_{init} is defined as $\llbracket \lambda(). t_{\text{init}} \rrbracket$
- $\text{final}_{\text{LEXA}}(\langle M \parallel H \parallel K \parallel [x_{\text{result}} \mapsto i] \parallel \text{halt} \rangle, i)$
- $\text{final}_{\text{SALT}}(\langle M \parallel H[L_{\text{sp}} \mapsto s :: i] \parallel R[\text{ip} \mapsto \ell, \text{sp} \mapsto L_{\text{sp}}^{[s :: i]}] \rangle, i)$ where $\hat{M}(\ell) = \text{halt}$

We first prove the first condition: to establish the relation $C_1 \sim C_2$, we need to show that its premises hold. The first premise, $\stackrel{\text{ins}}{\sim}$, holds: the term in LEXA is t_{init} , and the instruction sequence in SALT is defined as $\llbracket \lambda(). t_{\text{init}} \rrbracket$, so the relation is trivially established. The second premise, $\stackrel{\text{context}}{\sim}$, holds: the evaluation context and the local environment in LEXA are both empty, and that corresponds to having a single empty stack in the heap, with the stack pointer pointing to the base of the initial stack in SALT. The third premise, $\stackrel{\text{data}}{\sim}$, holds: the heaps are both empty. The fourth premise, $\stackrel{\text{code}}{\sim}$, holds: the code memory in LEXA and that in SALT are related by the definition of the program translation.

We now prove the second condition, that if $C_1 \sim C_2$, and C_1 is a final configuration, then C_2 is also a final configuration and their results are equal. Given that C_1 is a final configuration, the term in C_1 must be `halt`, and its environment must have exactly one binding of some number i . By the first premise of $C_1 \sim C_2$, the instruction sequence of C_2 must be a `halt` instruction. By the second premise of $C_1 \sim C_2$, the stack of C_2 must have i at the top. Therefore, C_2 is a final configuration, and the return value is i .

To prove the third condition, we do a case analysis on the reduction rules of LEXA. Most reduction rules of LEXA, for example ADD, involve computing the value of an expression and adding a binding of that value to the local environment. To prove these cases, we step through the translated instruction sequence in SALT and show that there is one more entry on the stack whose value matches the new binding in LEXA. We omit the details of these cases here. We also omit the LEAVE case, which follows the returning half of the HANDLE trampoline: control switches back to the parent stack, and the handler's stack is freed.

- Case APP: To show that $\sim^{\text{ins}}$ still holds after reducing C_1 to C'_1 , and C_2 to C'_2 , notice that the term in C'_1 is the body of the function, which corresponds to the instruction pointer in C'_2 pointing to the body of the function after the prologue. To show that $\sim^{\text{context}}$ still holds, notice that C'_1 has an extra activation frame and a new local environment consisting of arguments, which corresponds to C'_2 having an extra return address and arguments on the stack. The third and fourth premises, $\sim^{\text{data}}$ and $\sim^{\text{code}}$, still hold because the heap and the code memory are not changed.
- Case RET: Similar to APP, but in the opposite direction.
- Case HANDLE: We first prove the case for a general handler. To show that $\sim^{\text{ins}}$ still holds, notice that the term in C'_1 is the body of the handler, which corresponds to the instruction pointer in C'_2 pointing to the body of the handler. To show that $\sim^{\text{context}}$ still holds, notice that C'_1 has an extra activation frame and a handler frame, and a new local environment with two arguments. This corresponds to C'_2 having an extra return address on the old stack, and a new stack with the header frame whose entries match the handler frame in C'_1 . The new stack also has two arguments, which match the arguments in the local environment. The third and fourth premises, $\sim^{\text{data}}$ and $\sim^{\text{code}}$, still hold because the heap and the code memory are not changed.

For the tail-resumptive and abortive handlers, the proof is similar, except that C'_2 does not have a new stack. This difference is accounted for in $\sim^{\text{context}}$.

- Case RAISE: To show that $\overset{\text{ins}}{\sim}$ still holds, notice that the term in C'_1 is the body of the handler, which corresponds to the instruction pointer in C'_2 pointing to the body of the handler. To show that $\overset{\text{context}}{\sim}$ still holds, notice that C'_1 has the evaluation context cut at the handler frame, and a new environment is created with three arguments. This corresponds to C'_2 , where the set of stacks has been split into two and the stack pointer now points to the parent stack of the corresponding header frame. Since $\overset{\text{context}}{\sim}$ is defined structurally, we know that the shortened evaluation context and the reduced set of stacks are still related. To show that $\overset{\text{data}}{\sim}$ still holds, notice that C'_1 has a new single-element tuple containing the resumption. We need to show that the resumption is related to the other set of stacks in C'_2 . We again use the fact that $\overset{\text{context}}{\sim}$ is defined structurally to prove that. The fourth premise, $\overset{\text{code}}{\sim}$, still holds because the code memory is not changed.
- Case TAIL-RAISE: Similar to APP. We need to show that the operation functions used by C'_1 and C'_2 are related. This is true because the function location in the handler frame of C'_1 and in the header frame of C'_2 are related, according to $\overset{\text{context}}{\sim}$.
- Case ABORT-RAISE: Similar to RAISE, except that the captured evaluation context is discarded, which corresponds to discarding a subset of the stacks in the heap of C'_2 .
- Case RESUME: To show that $\overset{\text{ins}}{\sim}$ still holds, notice that the term in C'_1 comes from the last activation frame of the resumption. This corresponds to the new instruction pointer in C'_2 which comes from popping the last stack among the set of stacks that represents the resumption. The new term and the new instruction sequence are related by the definition of $\overset{\text{heap val}}{\sim}$. To show that $\overset{\text{context}}{\sim}$ still holds, notice that the resumption in C'_1 is related to a set of stacks in C'_2 according to $\overset{\text{heap val}}{\sim}$. Therefore, attaching the resumption to the evaluation context in C'_1 corresponds to joining the set of stacks in C'_2 and linking the exchanger of the header frame to the top of the last stack. To show that $\overset{\text{data}}{\sim}$ still holds, notice that both tuples that represent the resumption in C'_1 and C'_2 now contain `ns`. The fourth premise, $\overset{\text{code}}{\sim}$, still holds because the code memory is not changed.

Appendix B

Full Formalism for Zone LEXA

This appendix collects the complete formalism of SL and ZL from Sections 3.4 and 3.5.2: the source and runtime syntax, the operational semantics (including the administrative rules omitted from Figure 3.9), the full type system (of which Figures 3.10 and 3.11 show selected rules), and the parallel-let extension (syntax and operational semantics) of Section 3.6.5. The material reproduces the corresponding appendix of the technical report [36], adapted to the thesis page layout.

B.1 Source Program Syntax

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

B.2 Runtime Syntax

value $v ::= \dots \mid \text{lbl}(n) \mid \text{cont } K$
delimiter $\omega ::= \bowtie \mid \blacktriangleright$
frame $A ::= \square e \mid v \square \mid \text{raise } \square(e) \mid \text{raise } v(\square) \mid \text{resume } \square(e) \mid \text{resume } v(\square)$
stack $K ::= \epsilon \mid K :: A \mid K :: \omega_h^n$
zone $Z ::= \epsilon \mid Z :: A \mid Z :: \bowtie_h^n$
configuration $M ::= \langle K \parallel e \parallel m \rangle \mid \langle K \parallel K' \parallel n \parallel v \parallel m \rangle$

$$\begin{array}{c}
\boxed{\mathcal{L}(K)} \\
\mathcal{L}(\epsilon) = \emptyset \quad \mathcal{L}(K :: A) = \mathcal{L}(K) \quad \mathcal{L}(K :: \omega_h^n) = \mathcal{L}(K) \cup \{n\} \\
\boxed{\mathcal{L}(v)} \quad \boxed{\mathcal{L}(e)} \quad \boxed{\mathcal{L}(A)} \quad \boxed{\mathcal{L}(M)} \\
\mathcal{L}(x) = \emptyset \quad \mathcal{L}(() = \emptyset \quad \mathcal{L}(\lambda x.e) = \mathcal{L}(e) \quad \mathcal{L}(\text{lbl}(n)) = \emptyset \quad \mathcal{L}(\text{cont } K) = \mathcal{L}(K) \\
\mathcal{L}(e_1 e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \quad \mathcal{L}(\text{handle } \lambda x.e_1 \text{ with } \lambda(y,k).e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\
\mathcal{L}(\text{raise } e_1(e_2)) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \quad \mathcal{L}(\text{resume } e_1(e_2)) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\
\mathcal{L}(\Box e) = \mathcal{L}(e) \quad \mathcal{L}(v \Box) = \mathcal{L}(v) \\
\mathcal{L}(\text{raise } \Box(e)) = \mathcal{L}(e) \quad \mathcal{L}(\text{raise } v(\Box)) = \mathcal{L}(v) \\
\mathcal{L}(\text{resume } \Box(e)) = \mathcal{L}(e) \quad \mathcal{L}(\text{resume } v(\Box)) = \mathcal{L}(v) \\
\mathcal{L}(\langle K \parallel e \parallel m \rangle) = \mathcal{L}(K) \cup \mathcal{L}(e) \quad \mathcal{L}(\langle K \parallel K' \parallel n \parallel v \parallel m \rangle) = \mathcal{L}(K) \cup \mathcal{L}(K') \cup \mathcal{L}(v)
\end{array}$$

B.3 Operational Semantics

$$\begin{array}{ll}
\text{APP1 } \langle K \parallel e_1 e_2 \parallel m \rangle & \longrightarrow \langle K :: \Box e_2 \parallel e_1 \parallel m \rangle \\
\text{APP2 } \langle K \parallel v e \parallel m \rangle & \longrightarrow \langle K :: v \Box \parallel e \parallel m \rangle \\
\text{RAISE1 } \langle K \parallel \text{raise } e_1(e_2) \parallel m \rangle & \longrightarrow \langle K :: \text{raise } \Box(e_2) \parallel e_1 \parallel m \rangle \\
\text{RAISE2 } \langle K \parallel \text{raise } v(e) \parallel m \rangle & \longrightarrow \langle K :: \text{raise } v(\Box) \parallel e \parallel m \rangle \\
\text{RESUME1 } \langle K \parallel \text{resume } e_1(e_2) \parallel m \rangle & \longrightarrow \langle K :: \text{resume } \Box(e_2) \parallel e_1 \parallel m \rangle \\
\text{RESUME2 } \langle K \parallel \text{resume } v(e) \parallel m \rangle & \longrightarrow \langle K :: \text{resume } v(\Box) \parallel e \parallel m \rangle \\
\\
\text{BETA } \langle K \parallel (\lambda x.e) v \parallel m \rangle & \longrightarrow \langle K \parallel e \{v/x\} \parallel m \rangle \\
\text{ENTER } \langle K \parallel \text{handle } \lambda x.e \text{ with } h \parallel m \rangle & \longrightarrow \langle K :: \bowtie_h^m \parallel e \{ \text{lbl}(m)/x \} \parallel m+1 \rangle \\
\text{LEAVE1 } \langle K :: \omega_h^n \parallel v \parallel m \rangle & \longrightarrow \langle K \parallel v \parallel m \rangle \\
\text{LEAVE2 } \langle K :: A \parallel v \parallel m \rangle & \longrightarrow \langle K \parallel A[v] \parallel m \rangle \\
\text{RAISE } \langle K_1 :: \bowtie_h^n :: K_2 \parallel \text{raise } \text{lbl}(n)(v) \parallel m \rangle & \longrightarrow \langle K_1 \parallel \bowtie_h^n :: K_2 \parallel n \parallel v \parallel m \rangle \\
& \text{where } n \notin \mathcal{L}(K_2) \\
\text{RAISE } \langle K \parallel \text{raise } \text{lbl}(n)(v) \parallel m \rangle & \longrightarrow \langle K \parallel \epsilon \parallel n \parallel v \parallel m \rangle \\
\text{UNW-INTER-ZONE } \langle K :: \blacktriangleright_h^l :: Z \parallel K' \parallel n \parallel v \parallel m \rangle & \longrightarrow \langle K \parallel \blacktriangleright_h^l :: Z :: K' \parallel n \parallel v \parallel m \rangle \\
& \text{where } l \geq n \\
\text{UNW-INTRA-ZONE } \langle K :: \bowtie_h^n :: Z \parallel K' \parallel n \parallel v \parallel m \rangle & \longrightarrow \langle K \parallel \blacktriangleright_h^n :: Z :: K' \parallel n \parallel v \parallel m \rangle \\
\text{INVOKE } \langle K \parallel \omega_h^n :: K' \parallel n \parallel v \parallel m \rangle & \longrightarrow \\
& \langle K \parallel e \{v/y\} \{ \text{cont } \omega_h^n :: K'/k \} \parallel m \rangle \\
& \text{where } h = \lambda(y,k).e \\
\text{RESUME } \langle K \parallel \text{resume } (\text{cont } K')(v) \parallel m \rangle & \longrightarrow \langle K :: K' \parallel v \parallel m \rangle
\end{array}$$

B.4 Type System

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

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

$$\text{dom}(\epsilon) = \emptyset \quad \text{dom}(C \cup \{n \mapsto F\}) = \text{dom}(C) \cup \{n\}$$

$$\begin{array}{c}
\boxed{\Delta \vdash \tau} \quad \boxed{\Delta \vdash \eta} \quad \boxed{\Delta \vdash C} \\
\hline
\overline{\Delta \vdash \text{unit}} \quad \overline{\Delta \vdash \text{LBL}(\eta)} \quad \frac{\Delta, \Delta' \vdash C \quad \Delta, \Delta' \vdash \tau_1 \quad \Delta, \Delta' \vdash \tau_2}{\Delta \vdash \forall[\Delta' \mid C].\tau_1 \rightarrow \tau_2} \\
\hline
\overline{\Delta \vdash n} \quad \frac{\rho : \text{Label} \in \Delta}{\Delta \vdash \rho} \\
\hline
\overline{\Delta \vdash \epsilon} \quad \frac{\Delta \vdash C \quad \Delta \vdash \alpha}{\Delta \vdash C \cup \alpha} \quad \frac{\Delta \vdash C \quad \Delta \vdash \eta}{\Delta \vdash C \cup \{\eta \mapsto F\}} \\
\hline
\boxed{\Delta \mid \Gamma \vdash v : \tau} \\
\hline
\overline{\Delta \mid \Gamma \vdash () : \text{unit}} \quad \overline{\Delta \mid \Gamma \vdash x : \tau} \quad \frac{\Delta, \Delta' \mid \Gamma, x : \tau_1 \vdash e : \tau_2 \mid C}{\Delta \mid \Gamma \vdash \lambda x.e : \forall[\Delta' \mid C].\tau_1 \rightarrow \tau_2} \\
\hline
\overline{\Delta \mid \Gamma \vdash \text{lbl}(n) : \text{LBL}(n)} \quad \overline{\Delta \mid \Gamma \vdash \text{cont } K : \text{cont } [C].\tau_1 \rightarrow \tau_2} \\
\hline
\boxed{\Delta \mid \Gamma \vdash e : \tau \mid C} \\
\hline
\frac{\Delta \mid \Gamma \vdash e_1 : (\forall[\overline{\beta : \kappa} \mid C].\tau_1 \rightarrow \tau_2) \mid C \quad \Delta \mid \Gamma \vdash e_2 : \tau_1 \mid \{\bar{e}/\bar{\beta}\} \mid C \quad \Delta \vdash \bar{e} : \bar{\kappa}}{\Delta \mid \Gamma \vdash e_1 e_2 : \tau_2 \mid \{\bar{e}/\bar{\beta}\} \mid C} \\
\hline
\frac{\Delta, \rho : \text{Label} \mid \Gamma, x : \text{LBL}(\rho) \vdash e : \tau \mid C \cup \{\rho \mapsto F\} \quad \Delta \mid \Gamma \vdash h : F \# \tau \mid C \quad \Delta \vdash \tau}{\Delta \mid \Gamma \vdash \text{handle } \lambda x.e \text{ with } h : \tau \mid C} \\
\hline
\frac{\Delta \mid \Gamma \vdash e_1 : \text{LBL}(\eta) \mid C \quad C(\eta) = F \quad \Delta \mid \Gamma \vdash e_2 : \tau_1 \mid C \quad \mathbb{F}(F) = \tau_1 \rightarrow \tau_2}{\Delta \mid \Gamma \vdash \text{raise } e_1(e_2) : \tau_2 \mid C}
\end{array}$$

$$\begin{array}{c}
\frac{\Delta \mid \Gamma \vdash e_1 : (\text{cont}[C].\tau_2 \rightarrow \tau_{\text{ans}}) \mid C \quad \Delta \mid \Gamma \vdash e_2 : \tau_2 \mid C}{\Delta \mid \Gamma \vdash \text{resume } e_1(e_2) : \tau_{\text{ans}} \mid C} \\
\frac{\Delta \mid \Gamma \vdash e : \tau \mid C \quad C \subseteq C'}{\Delta \mid \Gamma \vdash e : \tau \mid C'} \\
\boxed{\Delta \mid \Gamma \vdash h : F \# \tau \mid C} \\
\frac{\Delta \mid \Gamma, y : \tau_1, k : \text{cont}[C].\tau_2 \rightarrow \tau_{\text{ans}} \vdash e : \tau_{\text{ans}} \mid C \quad \mathbb{F}(F) = \tau_1 \rightarrow \tau_2}{\Delta \mid \Gamma \vdash \lambda(y, k).e : F \# \tau_{\text{ans}} \mid C} \\
\frac{\Delta \mid \Gamma \vdash h : F \# \tau \mid C \quad C \subseteq C'}{\Delta \mid \Gamma \vdash h : F \# \tau \mid C'} \\
\boxed{\vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C} \\
\frac{\epsilon \mid \epsilon \vdash e : \tau_1 \mid C}{\vdash \Box e : (\forall [\epsilon \mid C].\tau_1 \rightarrow \tau_2) \rightsquigarrow \tau_2 \mid C} \\
\frac{\epsilon \mid \epsilon \vdash v : \forall [\epsilon \mid C].\tau_1 \rightarrow \tau_2}{\vdash v \Box : \tau_1 \rightsquigarrow \tau_2 \mid C} \\
\frac{C(n) = F \quad \epsilon \mid \epsilon \vdash e : \tau_1 \mid C \quad \mathbb{F}(F) = \tau_1 \rightarrow \tau_2}{\vdash \text{raise } \Box(e) : \text{LBL}(n) \rightsquigarrow \tau_2 \mid C} \\
\frac{C(n) = F \quad \epsilon \mid \epsilon \vdash v : \text{LBL}(n) \quad \mathbb{F}(F) = \tau_1 \rightarrow \tau_2}{\vdash \text{raise } v(\Box) : \tau_1 \rightsquigarrow \tau_2 \mid C} \\
\frac{\epsilon \mid \epsilon \vdash e : \tau_1 \mid C}{\vdash \text{resume } \Box(e) : (\text{cont}[C].\tau_1 \rightarrow \tau_2) \rightsquigarrow \tau_2 \mid C} \\
\frac{\epsilon \mid \epsilon \vdash v : \text{cont}[C].\tau_1 \rightarrow \tau_2}{\vdash \text{resume } v(\Box) : \tau_1 \rightsquigarrow \tau_2 \mid C} \\
\frac{\vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C \quad C \subseteq C'}{\vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C'} \\
\boxed{\vdash K : [C_1] \tau_1 \rightsquigarrow [C_2] \tau_2} \\
\frac{\vdash K : [C_2] \tau_2 \rightsquigarrow [C_3] \tau_3 \quad \vdash A : \tau_1 \rightsquigarrow \tau_2 \mid C_2}{\vdash K :: A : [C_2] \tau_1 \rightsquigarrow [C_3] \tau_3} \\
\frac{\vdash K : [C_1] \tau_1 \rightsquigarrow [C_2] \tau_2 \quad \epsilon \mid \epsilon \vdash h : F \# \tau_1 \mid C_1 \quad \forall l \in \text{dom}(C_1), l < n}{\vdash K :: \omega_h^n : [C_1 \cup \{n \mapsto F\}] \tau_1 \rightsquigarrow [C_2] \tau_2} \\
\frac{\vdash K : [C_1] \tau_1 \rightsquigarrow [C_2] \tau_2 \quad C'_1 \subseteq C_1 \quad C_2 \subseteq C'_2}{\vdash K : [C'_1] \tau_1 \rightsquigarrow [C'_2] \tau_2}
\end{array}$$

$$\begin{array}{c}
\boxed{\vdash M} \\
\hline
\vdash K : [C_1] \tau_1 \rightsquigarrow [\epsilon] \tau_2 \quad \epsilon \mid \epsilon \vdash e : \tau_1 \mid C_1 \quad \forall l \in \mathcal{L}(K) \cup \mathcal{L}(e), l \leq m \\
\vdash \langle K \parallel e \parallel m \rangle \\
\hline
\vdash K : [C_2] \tau_2 \rightsquigarrow [\epsilon] \tau_3 \quad \vdash K' : [C_1] \tau_1 \rightsquigarrow [C_2] \tau_2 \quad \forall l \in \mathcal{L}(K) \cup \mathcal{L}(K') \cup \mathcal{L}(v), l \leq m \\
n \notin \mathcal{L}(K') \quad C_2(n) = F \quad \epsilon \mid \epsilon \vdash v : \tau_{\text{arg}} \quad \mathbb{F}(F) = \tau_{\text{arg}} \rightarrow \tau_1 \\
\hline
\vdash \langle K \parallel K' \parallel n \parallel v \parallel m \rangle \\
\hline
\vdash K : [C_2] \tau_2 \rightsquigarrow [\epsilon] \tau_3 \quad \vdash K' : [C_1] \tau_1 \rightsquigarrow [C_2] \tau_2 \quad \forall l \in \mathcal{L}(K) \cup \mathcal{L}(K') \cup \mathcal{L}(v), l \leq m \\
K' = \omega_h^n :: K'' \quad C_1(n) = F \quad \epsilon \mid \epsilon \vdash v : \tau_{\text{arg}} \quad \mathbb{F}(F) = \tau_{\text{arg}} \rightarrow \tau_1 \\
\hline
\vdash \langle K \parallel K' \parallel n \parallel v \parallel m \rangle
\end{array}$$

B.5 Extension with Parallel-Let

expression $e ::= \dots \mid \text{par } x_1 = e_1, x_2 = e_2 \text{ in } e$
 configuration $M ::= \dots \mid \langle K \parallel x_1.x_2.e \parallel M_1 \parallel M_2 \parallel m \rangle$

$$\begin{array}{ll}
\text{P-FORK } \langle K \parallel \text{par } x_1 = e_1, x_2 = e_2 \text{ in } e \parallel m \rangle & \longrightarrow \\
& \langle K \parallel x_1.x_2.e \parallel \langle \epsilon \parallel e_1 \parallel m \rangle \parallel \langle \epsilon \parallel e_2 \parallel m \rangle \parallel m \rangle \\
\text{P-L } \langle K \parallel x_1.x_2.e \parallel M_1 \parallel M_2 \parallel m \rangle & \longrightarrow \langle K \parallel x_1.x_2.e \parallel M'_1 \parallel M_2 \parallel m \rangle \\
& \text{where } M_1 \rightarrow M'_1 \\
\text{P-R } \langle K \parallel x_1.x_2.e \parallel M_1 \parallel M_2 \parallel m \rangle & \longrightarrow \langle K \parallel x_1.x_2.e \parallel M_1 \parallel M'_2 \parallel m \rangle \\
& \text{where } M_2 \rightarrow M'_2 \\
\text{P-JOIN } \langle K \parallel x_1.x_2.e \parallel M_1 \parallel M_2 \parallel m \rangle & \longrightarrow \langle K \parallel e \{v_1/x_1\} \{v_2/x_2\} \parallel m \rangle \\
& \text{where } M_1 = \langle \epsilon \parallel v_1 \parallel m_1 \rangle \\
& \text{and } M_2 = \langle \epsilon \parallel v_2 \parallel m_2 \rangle
\end{array}$$

Both branches inherit the parent's label counter m , so labels generated in different branches may coincide, and P-JOIN discards the branch counters. This is harmless because the type system prevents effects from escaping a branch: labels generated in different branches are never compared against each other.

Appendix C

Metatheory Details for Zero LEXA

This appendix contains the metatheory deferred from Section 4.4: the subject-reduction development, the full definitions of the relations between SL^+ and TL constructs, and the simulation proof. The material reproduces the corresponding appendices of the technical report [32].

C.1 Subject Reduction

Before proving the simulation result, we first establish that runtime SL^+ configurations satisfy certain properties: capture sets are non-ambiguous and function types are non-frivolous. To this end, we prove the subject reduction theorem, which immediately yields the desired properties. We first present typing rules for SL^+ , which extends SL with runtime constructs. A new typing context Ψ is introduced, mapping run-time labels to their effect names. The typing context Θ is also updated to take into account run-time labels. The extended rule set includes the rules of Figures 4.11 and 4.12, with the typing context augmented by Ψ , and will not be repeated here. The new typing judgment forms and typing rules are given below, covering run-time labels L , reified continuation values $\text{cont } \#L^{(x,k) \Rightarrow t} K$, continuations K , evaluation contexts E , and configurations M .

Additional Typing Rules for SL^+ Runtime Constructs

$$\begin{aligned}\Theta &::= \epsilon \mid \alpha \mid \overline{l : F} \\ \Psi &::= \epsilon \mid \Psi, L : F\end{aligned}$$

$$\boxed{\Theta \mid \Sigma \mid \Psi \vdash \ell : F} \quad \boxed{\Theta \mid \Delta \mid \Sigma \mid \Psi \vdash T \text{ as } \Theta' \text{ unamb}}$$

...

$$\frac{(L : F \in \Psi) \vee (L : F \in \Theta)}{\Theta \mid \Sigma \mid \Psi \vdash L : F}$$

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

...

$$\frac{\begin{array}{c} \emptyset \mid \emptyset \mid \emptyset \mid x : \tau_1, k : \{\overline{L_{1F_1}}\} \text{ cont } \tau_2 \rightarrow \tau_{\text{ans}} \mid \overline{L_1 : F_1} \vdash t : \tau_{\text{ans}} \\ \overline{L_1 : F_1}, L_2 : F_2 \vdash K : \tau_2 \rightsquigarrow \tau_{\text{ans}} \quad \mathbb{F}(F_2) = \tau_1 \rightarrow \tau_2 \end{array}}{\emptyset \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \vdash \text{ cont } \{\overline{L_{1F_1}}\} \# L_2^{(x,k) \Rightarrow t} K : \{\overline{L_{1F_1}}\} \text{ cont } \tau_2 \rightarrow \tau_{\text{ans}}}$$

$$\boxed{\Psi \vdash K : \tau_1 \rightsquigarrow \tau_2}$$

...

$$\emptyset \vdash \epsilon : \tau \rightsquigarrow \tau$$

$$\frac{\Psi \vdash K : \tau_1 \rightsquigarrow \tau_2 \quad \emptyset \mid \emptyset \mid \emptyset \mid x : \tau_2 \mid \Psi \vdash t : \tau_3}{\Psi \vdash (\text{let } x = \square \text{ in } t) \cdot K : \tau_1 \rightsquigarrow \tau_3}$$

$$\frac{\begin{array}{c} \Psi, L : F \vdash K : \tau_1 \rightsquigarrow \tau_{\text{ans}} \quad \emptyset \mid \emptyset \mid \emptyset \mid x : \tau_{\text{ans}} \mid \Psi \vdash t_1 : \tau_2 \\ \emptyset \mid \emptyset \mid \emptyset \mid y : \tau_3, k : \{\Psi\} \text{ cont } \tau_4 \rightarrow \tau_{\text{ans}} \mid \Psi \vdash t_2 : \tau_{\text{ans}} \quad \mathbb{F}(F) = \tau_3 \rightarrow \tau_4 \end{array}}{\Psi \vdash (\text{let } x = \#L^{(y,k) \Rightarrow t_2} \square \text{ in } t_1) \cdot K : \tau_1 \rightsquigarrow \tau_2}$$

$$\boxed{\vdash E : \tau_1 \rightsquigarrow \tau_2 \mid \Psi}$$

...

$$\vdash \epsilon : \tau \rightsquigarrow \tau \mid \Psi$$

$$\frac{\vdash E : \tau_1 \rightsquigarrow \tau_2 \mid \Psi \quad \emptyset \mid \emptyset \mid \emptyset \mid x : \tau_3 \mid \Psi \vdash t : \tau_1}{\vdash E \cdot (\text{let } x = \square \text{ in } t) : \tau_3 \rightsquigarrow \tau_2 \mid \Psi}$$

$$\frac{\vdash E : \tau_1 \rightsquigarrow \tau_2 \mid \Psi \quad \emptyset \mid \emptyset \mid \emptyset \mid x : \tau_{\text{ans}} \mid \Psi \vdash t_1 : \tau_1 \quad \emptyset \mid \emptyset \mid \emptyset \mid y : \tau_3, k : \{\Psi\} \text{ cont } \tau_4 \rightarrow \tau_{\text{ans}} \mid \Psi \vdash t_2 : \tau_{\text{ans}} \quad \mathbb{F}(F) = \tau_3 \rightarrow \tau_4}{\vdash E \cdot (\text{let } x = \#L^{(y,k) \Rightarrow t_2} \square \text{ in } t_1) : \tau_{\text{ans}} \rightsquigarrow \tau_2 \mid \Psi, L : F}$$

$$\boxed{\vdash M}$$

$$\frac{\vdash E : \tau_1 \rightsquigarrow \tau_2 \mid \Psi \quad \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \mid \Psi \vdash t : \tau_1}{\vdash \langle E \parallel t \rangle}$$

$$\frac{\vdash E : \tau_2 \rightsquigarrow \tau_3 \mid \Psi \quad \Psi \vdash K : \tau_1 \rightsquigarrow \tau_2 \quad L : F \in \Psi \quad \mathbb{F}(F) = \tau_4 \rightarrow \tau_1 \quad \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \vdash v : \tau_4}{\vdash \langle E \parallel K \parallel L \parallel v \rangle}$$

$$\frac{\vdash E : \tau_2 \rightsquigarrow \tau_3 \mid \Psi \quad \Psi \vdash K : \tau_1 \rightsquigarrow \tau_2 \quad \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \vdash v : \tau_1}{\vdash \langle E \parallel K \parallel v \rangle}$$

Theorem. *If $\vdash M$ and $M \rightarrow M'$, then $\vdash M'$.*

Proof. By induction over the operational semantics. A standard substitution lemma is needed to prove the cases of application and handle expression. $\square$

Corollary C.1. *If $\emptyset \mid \emptyset \mid \emptyset \mid \emptyset \mid \Psi \vdash (\{\overline{L_{0F_0}}\} [\overline{\alpha}; \overline{\ell : F_2}] (\overline{x}) \Rightarrow t) [\overline{L_{0F_0}}; \overline{\overline{L_{1F_1}}}; \overline{\overline{L_{2F_2}}}] (\overline{v}) : \tau$, then $\overline{F_0}$ are unique and every set in $\overline{\overline{F_1}}$ is unique.*

Proof. $\overline{F_0}$ are unique due to the non-ambiguity of the function being called. $\overline{\overline{F_1}}$ are unique because each of them instantiates one of the $\overline{\alpha}$, which, due to the non-frivolous property, has to stand for part of the capture set of one of the function arguments. Since the function arguments all have non-ambiguous capture sets, any subset must be non-ambiguous, which means each of $\overline{\overline{F_1}}$ is unique. $\square$

C.2 Relating SL^+ and TL

SL ⁺ label	l	$::= \ell \mid L_F$
SL ⁺ capability	T	$::= \epsilon \mid \alpha \mid T, l$
SL ⁺ call-site instantiation	H	$::= T_0; \vec{T}; \vec{l}$
TL label index	ℓ	$::= \infty \mid \hat{i}$
TL capability index	α	$::= \infty \mid \hat{i}$
TL capability	T	$::= \epsilon \mid \alpha \mid T, \ell : F$
call-site metadata	H	$::= T_0; \vec{T}; \ell : F$
clue	C	$::= \langle \ell, F \rangle \mid \langle \alpha, F \rangle$
label promise	$\tilde{\Sigma}$	$::= \epsilon \mid \tilde{\Sigma}, \ell : F \sim \hat{i}$
capability promise	$\tilde{\Delta}$	$::= \epsilon \mid \tilde{\Delta}, \alpha \sim \hat{i}$
capture promise	$\tilde{\Theta}$	$::= \epsilon \mid \alpha \mid l : F$
promise	Π	$::= \tilde{\Theta}; \tilde{\Delta}; \tilde{\Sigma}$
evidence	Ξ	$::= \epsilon \mid \Xi, C \mapsto L$

C.2.1 Label and Capability Relation

An SL⁺ label l and a TL clue C are related through three rules. The first rule says that an SL⁺ label variable is related to a TL clue if such a relation appears in the label promise $\tilde{\Sigma}$. This rule is used to relate label parameters that are still open. The second rule says that an SL⁺ label (either variable or constant) is related to the infinite TL clue if such a label appears in the capture promise $\tilde{\Theta}$. This rule is used to relate captured label variables in a function, before the function is invoked.

These two rules use empty evidence Ξ because they are used in a context where the immediately enclosing function has not been invoked yet. In contrast, the last rule uses non-empty evidence Ξ and empty promises because it is used in a context where the immediately enclosing function has been invoked. The last rule relates an SL⁺ label constant to a TL clue if such a relation appears in the evidence Ξ . Notice that here ℓ stands for any clue index: ∞ , $\hat{i}$, or $\hat{i}$. When a function is invoked, all labels within the function body that were previously related through the first two rules are now related through the last rule.

Capabilities T and T are related using two types of rules. Rules on the first line are used to relate capabilities in a function before the function is invoked, so they use empty evidence Ξ . Rules on the second line are used to relate capabilities in a function after it

is invoked, so they use non-empty evidence Ξ . On the first line, the first rule relates the empty capabilities, the second rule relates an SL^+ capability parameter to a TL capability index of shape $\hat{i}$, the third rule relates a captured SL^+ capability variable to ∞ . The fourth rule is used when a capability variable that is captured by a function is substituted, but before the function is invoked. The final rule is a structural rule.

On the second line, the first two rules relate the base cases of T , and the last rule is a structural rule.

$$\begin{array}{c}
\boxed{l \stackrel{l}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Sigma}} C} \quad \boxed{T \stackrel{T}{\sim}_{\Xi|\Pi} T} \quad \boxed{H \sim_{\Xi|\Pi} H} \\
\\
\frac{\left(\ell : F \sim \hat{i} \right) \in \tilde{\Sigma}}{\ell \stackrel{l}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Sigma}} \langle \hat{i}, F \rangle} \quad \frac{(l : F) \in \tilde{\Theta}}{l \stackrel{l}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Sigma}} \langle \infty, F \rangle} \quad \frac{\Xi(\langle \ell, F \rangle) = L_F \quad F = F}{L_F \stackrel{l}{\sim}_{\Xi|\emptyset;\emptyset} \langle \ell, F \rangle} \\
\\
\hline
\frac{}{\epsilon \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \epsilon} \quad \frac{\left(\alpha \sim \hat{i} \right) \in \tilde{\Delta}}{\alpha \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \hat{i}} \quad \frac{\tilde{\Theta} = \alpha}{\alpha \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \infty} \quad \frac{\tilde{\Theta} = \vec{L}_F}{\vec{L}_F \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \infty} \quad \frac{T \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} T \quad l \stackrel{l}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Sigma}} \langle \ell, F \rangle}{T, l \stackrel{T}{\sim}_{\emptyset|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} T, \ell : F} \\
\\
\frac{}{\epsilon \stackrel{T}{\sim}_{\Xi|\emptyset;\emptyset} \epsilon} \quad \frac{\vec{L}_F^{(k)} \stackrel{l}{\sim}_{\Xi|\emptyset;\emptyset} \langle \alpha, \vec{F}^{(k)} \rangle \quad \vec{F}^{(k)} = \vec{F}^{(k)} \quad \text{for all } k}{\vec{L}_F \stackrel{T}{\sim}_{\Xi|\emptyset;\emptyset} \alpha} \quad \frac{T \stackrel{T}{\sim}_{\Xi|\emptyset;\emptyset} T \quad L_F \stackrel{l}{\sim}_{\Xi|\emptyset;\emptyset} \langle \ell, F \rangle}{T, L_F \stackrel{T}{\sim}_{\Xi|\emptyset;\emptyset} T, \ell : F} \\
\\
\hline
\frac{T_0 \stackrel{T}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} T_0 \quad T \stackrel{T}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} T \quad l \stackrel{l}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \langle \ell, F \rangle}{T_0; \vec{T}; \vec{l} \stackrel{\sim}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} T_0; \vec{T}; \ell : F}
\end{array}$$

C.2.2 Value Relation

We define `promise`, which is a function from $\mathbf{SL}^+$ constructs to promises, and `evidence`, which is a function from clues to $\mathbf{SL}^+$ run-time labels. For evidence to be a well-defined function, $\overline{F_0}$ and every set in $\overline{\overline{F_1}}$ need to be unique. This requirement on $\overline{F_0}$ and $\overline{\overline{F_1}}$ surfaces when proving the application case of the simulation proof, and is discharged by Corollary C.1.

$$\text{promise} \left(T_0; \vec{\alpha}; \ell : \vec{F} \right) = T_0; \left\{ \vec{\alpha}^{(i)} \sim \overset{\circ}{i} \mid i \right\}; \left\{ \ell : \vec{F}^{(i)} \sim \hat{i} \mid i \right\}$$

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

$$\boxed{v \overset{\text{val}}{\sim} v}$$

$$\frac{\frac{t \overset{\text{term}}{\sim} t \quad K \overset{\text{cont}}{\sim} K \quad \Xi = \text{evidence} \left(L_{0F_0}; \epsilon; \epsilon \right)}{\{T_0\} \left[\vec{\alpha}; \ell : \vec{F} \right] (\bar{x}) \Rightarrow t \overset{\text{val}}{\sim} (\bar{x}) \Rightarrow t} \quad \frac{t \overset{\text{term}}{\sim} t \quad K \overset{\text{cont}}{\sim} K \quad \Xi = \text{evidence} \left(L_{0F_0}; \epsilon; \epsilon \right)}{\text{cont} \left\{ L_{0F_0} \right\} \# L_F^{(x,k) \Rightarrow t} K \overset{\text{val}}{\sim} \text{cont} \#^{(x,k) \Rightarrow t} K}$$

C.2.3 Term Relation

$$\boxed{t \overset{\text{term}}{\sim} t \quad \Xi \mid \Pi}$$

$$\boxed{e \overset{\text{expr}}{\sim} e \quad \Xi \mid \Pi}$$

$$\frac{e \overset{\text{expr}}{\sim} e \quad t \overset{\text{term}}{\sim} t \quad \Xi \mid \Pi}{\text{let } x = e \text{ in } t \overset{\text{term}}{\sim} \text{let } x = e \text{ in } t \quad \Xi \mid \Pi}$$

$$\begin{array}{c}
\frac{l \overset{l}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Sigma}} C \quad v \overset{\text{val}}{\sim} v}{\text{raise } l(v) \overset{\text{expr}}{\sim}_{\Xi|\tilde{\Theta};\tilde{\Delta};\tilde{\Sigma}} \text{raise } C(v)} \\
\\
\frac{t_1 \overset{\text{term}}{\sim}_{\emptyset|\text{promise}(T_0;\epsilon;\{\ell:F\})} t_1 \quad t_2 \overset{\text{term}}{\sim}_{\emptyset|\text{promise}(T_0;\epsilon;\epsilon)} t_2 \quad T_0 \overset{T}{\sim}_{\Xi|\Pi} T}{\{T_0\} \text{handle } [\ell : F] \Rightarrow t_1 \text{ with } (x, k) \Rightarrow t_2 \overset{\text{expr}}{\sim}_{\Xi|\Pi} \text{handle}^T t_1 \text{ with } (x, k) \Rightarrow t_2} \\
\\
\frac{v_1 \overset{\text{val}}{\sim} v_1 \quad \vec{v}_2 \overset{\text{val}}{\sim} \vec{v}_2 \quad T_0; \vec{T}; \vec{l} \overset{}{\sim}_{\Xi|\Pi} H}{v_1 [T_0; \vec{T}; \vec{l}] (\vec{v}_2) \overset{\text{expr}}{\sim}_{\Xi|\Pi} (v_1(\vec{v}_2))^H} \\
\\
\frac{v_1 \overset{\text{val}}{\sim} v_1 \quad v_2 \overset{\text{val}}{\sim} v_2 \quad T \overset{T}{\sim}_{\Xi|\Pi} T}{\text{resume } v_1 [T] (v_2) \overset{\text{expr}}{\sim}_{\Xi|\Pi} \text{resume}^T v_1(v_2)}
\end{array}$$

C.2.4 Evaluation Context Relation

$$\boxed{\text{truthful}_{\Xi|H}(\Xi')}$$

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

$$\boxed{E \overset{\text{ctx}}{\sim}_{\Xi} E}$$

$$\frac{}{\epsilon \overset{\text{ctx}}{\sim}_{\emptyset} \epsilon}$$

$$\frac{E \overset{\text{ctx}}{\sim}_{\Xi} E \quad t \overset{\text{term}}{\sim}_{\Xi|\emptyset} t}{E \cdot (\text{let } x = \square \text{ in } t) \overset{\text{ctx}}{\sim}_{\Xi} E \cdot (\text{let } x = \square \text{ in } t)}$$

$$\begin{array}{c}
\frac{E \underset{\Xi}{\sim}^{\text{ctx}} E \quad t \underset{\Xi|\emptyset}{\sim}^{\text{term}} t \quad \text{truthful}_{\Xi|H}(\Xi')}{E \cdot (\text{let } x = \square \text{ in } t) \underset{\Xi'}{\sim}^{\text{ctx}} E \cdot (\text{let } x = \square^H \text{ in } t)} \\
\\
\frac{E \underset{\Xi}{\sim}^{\text{ctx}} E \quad t_2 \underset{\Xi|\emptyset}{\sim}^{\text{term}} t_2 \quad t_1 \underset{\Xi'|\emptyset}{\sim}^{\text{term}} t_1 \quad \text{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi') \quad \Xi' = \text{evidence} \left(\vec{L}_{0F_0}; \epsilon; \epsilon \right)}{E \cdot \left(\text{let } x = \left(\left\{ \vec{L}_{0F_0} \right\} \# L_F^{(y,k) \Rightarrow t_1} \square \right) \text{ in } t_2 \right) \underset{\Xi' \cup \{ \langle \vec{0}, F \rangle \mapsto L \}}{\sim}^{\text{ctx}} E \cdot \left(\text{let } x = (\#^{(y,k) \Rightarrow t_1} \square)^T \text{ in } t_2 \right)}
\end{array}$$

C.2.5 Continuation Relation

$$\begin{array}{c}
\boxed{K \underset{\Xi}{\sim}^{\text{cont}} K} \\
\\
\overline{\epsilon \underset{\Xi}{\sim}^{\text{cont}} \epsilon} \\
\\
\frac{K \underset{\Xi}{\sim}^{\text{cont}} K \quad t \underset{\Xi|\emptyset}{\sim}^{\text{term}} t}{(\text{let } x = \square \text{ in } t) \cdot K \underset{\Xi}{\sim}^{\text{cont}} (\text{let } x = \square \text{ in } t) \cdot K} \\
\\
\frac{K \underset{\Xi'}{\sim}^{\text{cont}} K \quad t \underset{\Xi|\emptyset}{\sim}^{\text{term}} t \quad \text{truthful}_{\Xi|H}(\Xi')}{(\text{let } x = \square \text{ in } t) \cdot K \underset{\Xi}{\sim}^{\text{cont}} (\text{let } x = \square^H \text{ in } t) \cdot K} \\
\\
\frac{K \underset{\Xi' \cup \{ \langle \vec{0}, F \rangle \mapsto L \}}{\sim}^{\text{cont}} K \quad t_2 \underset{\Xi|\emptyset}{\sim}^{\text{term}} t_2 \quad t_1 \underset{\Xi'|\emptyset}{\sim}^{\text{term}} t_1 \quad \text{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi') \quad \Xi' = \text{evidence} \left(\vec{L}_{0F_0}; \epsilon; \epsilon \right)}{(\text{let } x = \left(\left\{ \vec{L}_{0F_0} \right\} \# L_F^{(y,k) \Rightarrow t_1} \square \right) \text{ in } t_2) \cdot K \underset{\Xi}{\sim}^{\text{cont}} (\text{let } x = (\#^{(y,k) \Rightarrow t_1} \square)^T \text{ in } t_2) \cdot K}
\end{array}$$

C.2.6 Configuration Relation

$$\boxed{M \sim M}$$

$$\begin{array}{c}
\frac{E \stackrel{\text{ctx}}{\sim}_{\Xi} E \quad t \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t}{\langle E \parallel t \rangle \sim \langle E \parallel t \rangle} \\
\\
\frac{E \stackrel{\text{ctx}}{\sim}_{\Xi} E \quad K \stackrel{\text{cont}}{\sim}_{\Xi} K \quad L \stackrel{l}{\sim}_{\Xi|\emptyset} C \quad v \stackrel{\text{val}}{\sim} v}{\langle E \parallel K \parallel L \parallel v \rangle \sim \langle E \parallel K \parallel C \parallel v \rangle} \\
\\
\frac{E \stackrel{\text{ctx}}{\sim}_{\Xi} E \quad K \stackrel{\text{cont}}{\sim}_{\Xi} K \quad v \stackrel{\text{val}}{\sim} v}{\langle E \parallel K \parallel v \rangle \sim \langle E \parallel K \parallel v \rangle}
\end{array}$$

C.3 Simulation Proof

Theorem. *If $\vdash M$, $M \sim M$, and $M \rightarrow M'$, then there exists M' such that $M \rightarrow^* M'$ and $M' \sim M'$.*

Proof. By induction on $M \rightarrow M'$. We show the cases for the interesting rules. The other cases are straightforward.

Case 1. APP: in the following diagram, $\rightarrow_1$ and $\sim_1$ are given in the premise. $\rightarrow_2$ is given by the operational semantics of TL, and we want to prove that $\sim_2$ holds.

$$\begin{array}{ccc}
\langle E \parallel \text{let } x = (\{\overline{L_{0F_0}}\} [\vec{\alpha}; \ell : \vec{F}] (\vec{x}) \Rightarrow t_1) [L_{0F_0}; L_{1F_1}; L_{2F_2}] (\vec{v}) \text{ in } t_2 \rangle & \xrightarrow{1} & \langle E \cdot (\text{let } x = \square \text{ in } t_2) \parallel t_1 [\overline{\alpha \mapsto L_{1F_1}}, \overline{\ell \mapsto L_{2F_2}}, \overline{x \mapsto v}] \rangle \\
\sim_1 & & \sim_2 \\
\langle E \parallel \text{let } x = ((\vec{x}) \Rightarrow t_1)^H (\vec{v}) \text{ in } t_2 \rangle & \xrightarrow{2} & \langle E \cdot (\text{let } x = \square^H \text{ in } t_2) \parallel t_1 [\overline{x \mapsto v}] \rangle
\end{array}$$

By inversion on $\sim_1$, we are given the following derivation.

$$\begin{array}{c}
\begin{array}{c} \overbrace{\quad\quad\quad}^4 \\ \begin{array}{c} \text{term} \\ t_1 \sim_{\emptyset | \text{promise}(\overline{L_{0F_0}}; \vec{\alpha}; \ell; \vec{F})} t_1 \end{array} \end{array} \\
\hline
\begin{array}{c} \overbrace{\quad\quad\quad}^5 \quad \overbrace{\quad\quad\quad}^3 \\ \begin{array}{c} \vec{v} \text{ val} \vec{v} \quad L_{0F_0}; L_{1F_1}; L_{2F_2} \sim_{\Xi|\emptyset}^H \end{array} \end{array} \\
\hline
\begin{array}{c} \overbrace{\quad\quad\quad}^2 \\ t_2 \text{ term} \sim_{\Xi|\emptyset} t_2 \end{array} \\
\hline
\begin{array}{c} \overbrace{\quad\quad\quad}^1 \\ E \text{ ctx} \sim_{\Xi} E \end{array} \\
\hline
\begin{array}{c} \left(\{\overline{L_{0F_0}}\} [\vec{\alpha}; \ell : \vec{F}] (\vec{x}) \Rightarrow t_1 \right) [L_{0F_0}; L_{1F_1}; L_{2F_2}] (\vec{v}) \sim_{\Xi|\emptyset}^{\text{expr}} ((\vec{x}) \Rightarrow t_1)^H (\vec{v}) \\ \text{let } x = (\{\overline{L_{0F_0}}\} [\vec{\alpha}; \ell : \vec{F}] (\vec{x}) \Rightarrow t_1) [L_{0F_0}; L_{1F_1}; L_{2F_2}] (\vec{v}) \text{ in } t_2 \sim_{\Xi|\emptyset}^{\text{term}} \text{let } x = ((\vec{x}) \Rightarrow t_1)^H (\vec{v}) \text{ in } t_2 \end{array} \\
\hline
\langle E \parallel \text{let } x = (\{\overline{L_{0F_0}}\} [\vec{\alpha}; \ell : \vec{F}] (\vec{x}) \Rightarrow t_1) [L_{0F_0}; L_{1F_1}; L_{2F_2}] (\vec{v}) \text{ in } t_2 \rangle \sim \langle E \parallel \text{let } x = ((\vec{x}) \Rightarrow t_1)^H (\vec{v}) \text{ in } t_2 \rangle
\end{array}$$

We want to show that there exists Ξ' such that the following derivation for $\sim_2$ holds.

$$\begin{array}{c}
\begin{array}{c} \overbrace{\quad\quad\quad}^1 \\ E \text{ ctx} \sim_{\Xi} E \end{array} \quad \begin{array}{c} \overbrace{\quad\quad\quad}^2 \\ t_2 \text{ term} \sim_{\Xi|\emptyset} t_2 \end{array} \quad \begin{array}{c} \overbrace{\quad\quad\quad}^3 \\ \text{truthful}_{\Xi|H}(\Xi') \end{array} \\
\hline
\begin{array}{c} \overbrace{\quad\quad\quad}^4 \\ t_1 [\overline{\alpha \mapsto L_{1F_1}}, \overline{\ell \mapsto L_{2F_2}}, \overline{x \mapsto v}] \sim_{\Xi'|\emptyset}^{\text{term}} t_1 [\overline{x \mapsto v}] \end{array} \\
\hline
\begin{array}{c} E \cdot (\text{let } x = \square \text{ in } t_2) \sim_{\Xi'}^{\text{ctx}} E \cdot (\text{let } x = \square^H \text{ in } t_2) \end{array} \\
\hline
\langle E \cdot (\text{let } x = \square \text{ in } t_2) \parallel t_1 [\overline{\alpha \mapsto L_{1F_1}}, \overline{\ell \mapsto L_{2F_2}}, \overline{x \mapsto v}] \rangle \sim \langle E \cdot (\text{let } x = \square^H \text{ in } t_2) \parallel t_1 [\overline{x \mapsto v}] \rangle
\end{array}$$

We choose $\Xi' = \text{evidence} \left(\vec{L}_{0F_0}; \vec{L}_{1F_1}; \vec{L}_{2F_2} \right)$. For Ξ' to be a well-defined function, we require $\overline{F_0}$ and every set in $\overline{\overline{F_1}}$ to be unique. This is derived by applying Corollary C.1 to the well-typedness of M . The named premises in the above derivation are proved as follows, using the premises from the given derivation.

1. $1 \rightarrow 1$
2. $2 \rightarrow 2$
3. $3 \rightarrow 3$ by the context plumbing lemma
4. $4 \wedge 5 \rightarrow 4$ by the term plumbing lemma and the substitution lemma

Case 2. HANDLE:

$$\begin{aligned}
\langle E \parallel \text{let } x = (\{\overline{L_{0F_0}}\} \text{ handle } [\ell : F] \Rightarrow t_1 \text{ with } (y, k) \Rightarrow t_2) \text{ in } t_3 \rangle &\longrightarrow_1 \langle E \cdot (\text{let } x = (\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square) \text{ in } t_3) \parallel t_1 [\ell \mapsto L] \rangle \text{ where } L \text{ is fresh} \\
&\sim_1 \\
\langle E \parallel \text{let } x = (\text{handle}^T t_1 \text{ with } (y, k) \Rightarrow t_2) \text{ in } t_3 \rangle &\longrightarrow_2 \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_3) \parallel t_1 \rangle
\end{aligned}$$

By inversion on $\sim_1$, we are given

$$\begin{array}{c}
\overline{\overbrace{E}^1} \xrightarrow{\text{ctx}} \overline{\overbrace{E}^1} \xrightarrow{\Xi} \overline{\overbrace{E}^1} \quad \overline{\overbrace{t_1}^2} \xrightarrow{\text{term}} \overline{\overbrace{t_1}^2} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_1}^2} \quad \overline{\overbrace{t_2}^3} \xrightarrow{\text{term}} \overline{\overbrace{t_2}^3} \xrightarrow{\Xi|\text{promise}(\overline{L_{0F_0}}; \epsilon; \{\ell:F\})} \overline{\overbrace{t_2}^3} \quad \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{T} \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{L_{0F_0}}^4} \quad \overline{\overbrace{t_3}^5} \xrightarrow{\text{term}} \overline{\overbrace{t_3}^5} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_3}^5} \\
\hline
\overline{\overbrace{E}^1} \xrightarrow{\text{ctx}} \overline{\overbrace{E}^1} \xrightarrow{\Xi} \overline{\overbrace{E}^1} \quad \overline{\overbrace{t_1}^2} \xrightarrow{\text{term}} \overline{\overbrace{t_1}^2} \xrightarrow{\Xi|\text{promise}(\overline{L_{0F_0}}; \epsilon; \{\ell:F\})} \overline{\overbrace{t_1}^2} \quad \overline{\overbrace{t_2}^3} \xrightarrow{\text{term}} \overline{\overbrace{t_2}^3} \xrightarrow{\Xi|\text{promise}(\overline{L_{0F_0}}; \epsilon; \{\ell:F\})} \overline{\overbrace{t_2}^3} \quad \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{T} \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{L_{0F_0}}^4} \quad \overline{\overbrace{t_3}^5} \xrightarrow{\text{term}} \overline{\overbrace{t_3}^5} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_3}^5} \\
\hline
\overline{\overbrace{E}^1} \xrightarrow{\text{ctx}} \overline{\overbrace{E}^1} \xrightarrow{\Xi} \overline{\overbrace{E}^1} \quad \overline{\overbrace{t_1}^2} \xrightarrow{\text{term}} \overline{\overbrace{t_1}^2} \xrightarrow{\Xi|\text{promise}(\overline{L_{0F_0}}; \epsilon; \{\ell:F\})} \overline{\overbrace{t_1}^2} \quad \overline{\overbrace{t_2}^3} \xrightarrow{\text{term}} \overline{\overbrace{t_2}^3} \xrightarrow{\Xi|\text{promise}(\overline{L_{0F_0}}; \epsilon; \{\ell:F\})} \overline{\overbrace{t_2}^3} \quad \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{T} \overline{\overbrace{L_{0F_0}}^4} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{L_{0F_0}}^4} \quad \overline{\overbrace{t_3}^5} \xrightarrow{\text{term}} \overline{\overbrace{t_3}^5} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_3}^5} \\
\hline
\langle E \parallel \text{let } x = (\{\overline{L_{0F_0}}\} \text{ handle } [\ell : F] \Rightarrow t_1 \text{ with } (y, k) \Rightarrow t_2) \text{ in } t_3 \rangle \sim \langle E \parallel \text{let } x = (\text{handle}^T t_1 \text{ with } (y, k) \Rightarrow t_2) \text{ in } t_3 \rangle
\end{array}$$

We want to show that the following derivation holds.

$$\begin{array}{c}
\overline{\overbrace{E}^1} \xrightarrow{\text{ctx}} \overline{\overbrace{E}^1} \xrightarrow{\Xi} \overline{\overbrace{E}^1} \quad \overline{\overbrace{t_3}^2} \xrightarrow{\text{term}} \overline{\overbrace{t_3}^2} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_3}^2} \quad \overline{\overbrace{t_2}^3} \xrightarrow{\text{term}} \overline{\overbrace{t_2}^3} \xrightarrow{\Xi'|\emptyset} \overline{\overbrace{t_2}^3} \quad \overline{\text{truthful}_{\Xi|T; \epsilon; \epsilon}(\Xi')} \quad \Xi' = \text{evidence}(\overline{L_{0F_0}}; \epsilon; \epsilon) \quad \overline{\overbrace{t_1}^5} \xrightarrow{\text{term}} \overline{\overbrace{t_1}^5} \xrightarrow{\Xi' \cup \{\langle \emptyset, F \rangle \mapsto L\}|\emptyset} \overline{\overbrace{t_1}^5} \\
\hline
\overline{\overbrace{E}^1} \xrightarrow{\text{ctx}} \overline{\overbrace{E}^1} \xrightarrow{\Xi} \overline{\overbrace{E}^1} \quad \overline{\overbrace{t_3}^2} \xrightarrow{\text{term}} \overline{\overbrace{t_3}^2} \xrightarrow{\Xi|\emptyset} \overline{\overbrace{t_3}^2} \quad \overline{\overbrace{t_2}^3} \xrightarrow{\text{term}} \overline{\overbrace{t_2}^3} \xrightarrow{\Xi'|\emptyset} \overline{\overbrace{t_2}^3} \quad \overline{\text{truthful}_{\Xi|T; \epsilon; \epsilon}(\Xi')} \quad \Xi' = \text{evidence}(\overline{L_{0F_0}}; \epsilon; \epsilon) \quad \overline{\overbrace{t_1}^5} \xrightarrow{\text{term}} \overline{\overbrace{t_1}^5} \xrightarrow{\Xi' \cup \{\langle \emptyset, F \rangle \mapsto L\}|\emptyset} \overline{\overbrace{t_1}^5} \\
\hline
\langle E \cdot (\text{let } x = (\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square) \text{ in } t_3) \parallel t_1 [\ell \mapsto L] \rangle \sim \langle E \cdot (\text{let } x = (\#^{(y,k) \Rightarrow t_2} \square)^T \text{ in } t_3) \parallel t_1 \rangle
\end{array}$$

1. $1 \rightarrow 1$
2. $5 \rightarrow 2$
3. $3 \rightarrow 3$ by the term plumbing lemma

4. $4 \rightarrow 4$ by the context plumbing lemma
5. $2 \rightarrow 5$ by the substitution lemma and the term plumbing lemma

Case 3. UNW-LET:

$$\begin{aligned}
\langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel L \parallel v \rangle &\longrightarrow \langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel L \parallel v \rangle \\
&\sim \\
\langle E \cdot (\text{let } x = \square^H \text{ in } t) \parallel K \parallel C \parallel v \rangle &\longrightarrow \langle E \parallel (\text{let } x = \square^H \text{ in } t) \cdot K \parallel \text{hopper}_H(C) \parallel v \rangle
\end{aligned}$$

We are given

$$\frac{
\frac{
\overbrace{E \stackrel{\text{ctx}}{\sim} E}^1
\quad
\overbrace{t \stackrel{\text{term}}{\sim} t}_{\Xi|\emptyset}^2
\quad
\overbrace{\text{truthful}_{\Xi|H}(\Xi')}^3
}{
E \cdot (\text{let } x = \square \text{ in } t) \stackrel{\text{ctx}}{\sim}_{\Xi'} E \cdot (\text{let } x = \square^H \text{ in } t)
}
\quad
\overbrace{K \stackrel{\text{cont}}{\sim}_{\Xi'} K}^4
\quad
\overbrace{L \stackrel{l}{\sim}_{\Xi'|\emptyset} C}^5
\quad
\overbrace{v \stackrel{\text{val}}{\sim} v}^6
}{
\langle E \cdot (\text{let } x = \square \text{ in } t) \parallel K \parallel L \parallel v \rangle \sim \langle E \cdot (\text{let } x = \square^H \text{ in } t) \parallel K \parallel C \parallel v \rangle
}$$

We want to show that the following derivation holds.

$$\frac{
\overbrace{E \stackrel{\text{ctx}}{\sim} E}^1
\quad
\frac{
\overbrace{K \stackrel{\text{cont}}{\sim}_{\Xi'} K}^2
\quad
\overbrace{t \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t}^3
\quad
\overbrace{\text{truthful}_{\Xi|H}(\Xi')}^4
}{
(\text{let } x = \square \text{ in } t) \cdot K \stackrel{\text{cont}}{\sim}_{\Xi} (\text{let } x = \square^H \text{ in } t) \cdot K
}
\quad
\overbrace{L \stackrel{l}{\sim}_{\Xi|\emptyset} \text{hopper}_H(C)}^5
\quad
\overbrace{v \stackrel{\text{val}}{\sim} v}^6
}{
\langle E \parallel (\text{let } x = \square \text{ in } t) \cdot K \parallel L \parallel v \rangle \sim \langle E \parallel (\text{let } x = \square^H \text{ in } t) \cdot K \parallel \text{hopper}_H(C) \parallel v \rangle
}$$

1. $1 \rightarrow 1$
2. $4 \rightarrow 2$
3. $2 \rightarrow 3$
4. $3 \rightarrow 4$
5. $3 \wedge 5 \rightarrow 5$ by the definition of truthful
6. $6 \rightarrow 6$

Case 4. FOUND:

$$\begin{aligned}
& \left\langle E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \parallel K \parallel L \parallel v \right\rangle \longrightarrow \left\langle E \cdot \left(\text{let } x = \square \text{ in } t_1 \right) \parallel t_2 \left[y \mapsto v, k \mapsto \text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right] \right\rangle \\
& \quad \sim \left\langle E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right) \parallel K \parallel \langle \hat{0}, F \rangle \parallel v \right\rangle \longrightarrow \left\langle E \cdot \left(\text{let } x = \square^{T;\epsilon;\epsilon} \text{ in } t_1 \right) \parallel t_2 \left[y \mapsto v, k \mapsto \text{cont } \#^{(y,k) \Rightarrow t_2} K \right] \right\rangle
\end{aligned}$$

We are given

$$\frac{
\begin{array}{c}
\overbrace{E \stackrel{\text{ctx}}{\sim} E}^1 \quad \overbrace{t_1 \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t_1}^2 \quad \overbrace{t_2 \stackrel{\text{term}}{\sim}_{\Xi'|\emptyset} t_2}^3 \quad \overbrace{\text{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi')}^4 \quad \Xi' = \text{evidence} \left(\vec{L_{0F_0}}; \epsilon; \epsilon \right) \\
\hline
E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \stackrel{\text{ctx}}{\sim}_{\Xi' \cup \{\langle \hat{0}, F \rangle \mapsto L\}} E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right)
\end{array}
\quad
\begin{array}{c}
\overbrace{K \stackrel{\text{cont}}{\sim}_{\Xi' \cup \{\langle \hat{0}, F \rangle \mapsto L\}} K}^5 \quad \overbrace{L \stackrel{l}{\sim}_{\Xi' \cup \{\langle \hat{0}, F \rangle \mapsto L\}|\emptyset} \langle \hat{0}, F \rangle}^6 \quad \overbrace{v \stackrel{\text{val}}{\sim} v}^7 \\
\hline
\left\langle E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \parallel K \parallel L \parallel v \right\rangle \sim \left\langle E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right) \parallel K \parallel \langle \hat{0}, F \rangle \parallel v \right\rangle
\end{array}$$

We want to show that the following derivation holds.

$$\frac{
\begin{array}{c}
\overbrace{E \stackrel{\text{ctx}}{\sim} E}^1 \quad \overbrace{t_1 \stackrel{\text{term}}{\sim}_{\Xi|\emptyset} t_1}^2 \quad \overbrace{\text{truthful}_{\Xi|T;\epsilon;\epsilon}(\Xi')}^3 \quad \overbrace{t_2 \stackrel{\text{term}}{\sim}_{\Xi'|\emptyset} t_2}^4 \quad \overbrace{v \stackrel{\text{val}}{\sim} v}^5 \quad \overbrace{K \stackrel{\text{cont}}{\sim}_{\Xi'' \cup \{\langle \hat{0}, F \rangle \mapsto L\}} K}^6 \quad \overbrace{\Xi'' = \text{evidence} \left(\vec{L_{0F_0}}; \epsilon; \epsilon \right)}^7 \\
\hline
E \cdot \left(\text{let } x = \square \text{ in } t_1 \right) \stackrel{\text{ctx}}{\sim}_{\Xi'} E \cdot \left(\text{let } x = \square^{T;\epsilon;\epsilon} \text{ in } t_1 \right) \quad t_2 \left[y \mapsto v, k \mapsto \text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right] \stackrel{\text{term}}{\sim}_{\Xi'|\emptyset} t_2 \left[y \mapsto v, k \mapsto \text{cont } \#^{(y,k) \Rightarrow t_2} K \right]
\end{array}
\quad
\begin{array}{c}
\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \stackrel{\text{val}}{\sim} \text{cont } \#^{(y,k) \Rightarrow t_2} K \\
\hline
\left\langle E \cdot \left(\text{let } x = \square \text{ in } t_1 \right) \parallel t_2 \left[y \mapsto v, k \mapsto \text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right] \right\rangle \sim \left\langle E \cdot \left(\text{let } x = \square^{T;\epsilon;\epsilon} \text{ in } t_1 \right) \parallel t_2 \left[y \mapsto v, k \mapsto \text{cont } \#^{(y,k) \Rightarrow t_2} K \right] \right\rangle
\end{array}$$

1. $1 \rightarrow 1$
2. $2 \rightarrow 2$
3. $4 \rightarrow 3$
4. $3 \rightarrow 4$

5. $7 \rightarrow 5$

6. $3 \rightarrow 6$, notice that $\Xi' = \Xi''$

7. $5 \rightarrow 7$, notice that $\Xi' = \Xi''$

Case 5. RESUME:

$$\begin{aligned}
& \langle E \parallel \text{let } x = \text{resume} \left(\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right) [\overline{L_{0F_0}}] (v) \text{ in } t_1 \rangle \longrightarrow \langle E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \parallel K \parallel v \rangle \\
& \quad \sim \quad \langle E \parallel \text{let } x = \text{resume}^T \left(\text{cont } \#^{(y,k) \Rightarrow t_2} K \right) (v) \text{ in } t_1 \rangle \longrightarrow \langle E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right) \parallel K \parallel v \rangle
\end{aligned}$$

We are given

$$\begin{array}{c}
\begin{array}{c} \overbrace{t_2 \xrightarrow{\text{term}} t_2}^2 \\ \Xi' | \emptyset \end{array} \quad \begin{array}{c} \overbrace{K \xrightarrow{\text{cont}} K}^3 \\ \Xi' \cup \{ \langle \hat{0}, F \rangle \mapsto L \} \end{array} \quad \Xi' = \text{evidence} \left(L_{0F_0}^{\rightarrow}; \epsilon; \epsilon \right) \quad \begin{array}{c} \overbrace{v \xrightarrow{\text{val}} v}^4 \\ \Xi' | \emptyset \end{array} \quad \begin{array}{c} \overbrace{L_{0F_0} \xrightarrow{T} T}^5 \\ \Xi' | \emptyset \end{array} \quad \begin{array}{c} \overbrace{t_1 \xrightarrow{\text{term}} t_1}^6 \\ \Xi' | \emptyset \end{array} \\
\hline
\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \xrightarrow{\text{val}} \text{cont } \#^{(y,k) \Rightarrow t_2} K \\
\hline
\text{resume} \left(\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right) [\overline{L_{0F_0}}] (v) \xrightarrow{\text{expr}} \text{resume}^T \left(\text{cont } \#^{(y,k) \Rightarrow t_2} K \right) (v) \\
\hline
\begin{array}{c} \overbrace{E \xrightarrow{\text{ctx}} E}^1 \\ \Xi \end{array} \quad \text{let } x = \text{resume} \left(\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right) [\overline{L_{0F_0}}] (v) \text{ in } t_1 \xrightarrow{\text{term}} \text{let } x = \text{resume}^T \left(\text{cont } \#^{(y,k) \Rightarrow t_2} K \right) (v) \text{ in } t_1 \\
\hline
\langle E \parallel \text{let } x = \text{resume} \left(\text{cont } \{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} K \right) [\overline{L_{0F_0}}] (v) \text{ in } t_1 \rangle \sim \langle E \parallel \text{let } x = \text{resume}^T \left(\text{cont } \#^{(y,k) \Rightarrow t_2} K \right) (v) \text{ in } t_1 \rangle
\end{array}$$

We want to show that the following derivation holds.

$$\begin{array}{c}
\begin{array}{c} \overbrace{E \xrightarrow{\text{ctx}} E}^1 \\ \Xi \end{array} \quad \begin{array}{c} \overbrace{t_1 \xrightarrow{\text{term}} t_1}^2 \\ \Xi' | \emptyset \end{array} \quad \begin{array}{c} \overbrace{t_2 \xrightarrow{\text{term}} t_2}^3 \\ \Xi'' | \emptyset \end{array} \quad \overbrace{\text{truthful}_{\Xi | T; \epsilon; \epsilon}(\Xi'')}^4 \quad \Xi'' = \text{evidence} \left(L_{0F_0}^{\rightarrow}; \epsilon; \epsilon \right) \quad \begin{array}{c} \overbrace{K \xrightarrow{\text{cont}} K}^5 \\ \Xi' \cup \{ \langle \hat{0}, F \rangle \mapsto L \} \end{array} \quad \begin{array}{c} \overbrace{v \xrightarrow{\text{val}} v}^6 \\ \Xi' | \emptyset \end{array} \\
\hline
E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \xrightarrow{\text{ctx}} E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right) \\
\hline
\langle E \cdot \left(\text{let } x = \left(\{\overline{L_{0F_0}}\} \# L_F^{(y,k) \Rightarrow t_2} \square \right) \text{ in } t_1 \right) \parallel K \parallel v \rangle \sim \langle E \cdot \left(\text{let } x = \left(\#^{(y,k) \Rightarrow t_2} \square \right)^T \text{ in } t_1 \right) \parallel K \parallel v \rangle
\end{array}$$

We choose Ξ' to be the one in the given derivation.

1. $1 \rightarrow 1$
2. $6 \rightarrow 2$
3. $2 \rightarrow 3$, notice that $\Xi' = \Xi''$
4. $5 \rightarrow 4$ by the context plumbing lemma
5. $3 \rightarrow 5$
6. $4 \rightarrow 6$

□