Dynamic Wind for Effect Handlers

DAVID VOIGT, University of Tübingen, Germany
PHILIPP SCHUSTER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Effect handlers offer an attractive way of abstracting over effectful computation. Moreover, languages with effect handlers usually statically track effects, which ensures the user is aware of all side effects different parts of a program might have. Similarly to exception handlers, effect handlers discharge effects by locally defining their behavior. In contrast to exception handlers, they allow for resuming computation, possibly later and possibly multiple times. In this paper we present a design, formalization, and implementation for a variant of dynamic wind that integrates well with lexical effect handlers. It has well-defined semantics in the presence of arbitrary control effects in arbitrary places. Specifically, the behavior of capturing and resuming continuations in the pre- or postlude is well-defined and respects resource bracketing. We demonstrate how these features can be used to express backtracking of external state and finalization of external resources.

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

Additional Key Words and Phrases: control flow, lexical effect handlers, resource management, finalization, continuations, multiple resumption, effect systems, capabilities

ACM Reference Format:
David Voigt, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. Dynamic Wind for Effect Handlers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 377 (October 2025), 41 pages. https://doi.org/10.1145/3763155

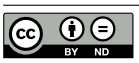
1 Introduction

Practical programs necessarily interact with the real world — they have *side effects*. Moreover, parts of a program interact with other parts of the program in an effectful way, for example through mutating a shared reference cell, or by throwing an exception. Due to their non-local nature, side effects make it harder to reason about programs: one has to take into account the context in which a program runs in order to make sense of its behavior.

A *type and effect system* [Lucassen and Gifford 1988] tracks not only the types of values in a program, but also the side effects of computations. This way, programmers immediately see which side effects each part of a program can have. Effect safety means that a program at runtime indeed has at most those effects assigned to it by the static effect system. This way, when trying to understand a program, it is clear which parts of the context are relevant and which are not.

Effect handlers are a language feature that allow for the local encapsulation of side effects and [Plotkin and Pretnar 2009, 2013]. In analogy to exception handlers, effect handlers locally discharge an effect, which can therefore not affect the program outside the handler — they delimit the context of an effect.

Authors' Contact Information: David Voigt, University of Tübingen, Germany, david.voigt@uni-tuebingen.de; Philipp Schuster, University of Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, University of Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de.



ACM 2475-1421/2025/10-ART377
https://doi.org/10.1145/3763155

In contrast to exception handlers, however, effect handlers can resume computations, potentially at a later time and even multiple times. Therefore, they make it possible to express and abstract over advanced patterns of control flow. In other words, they allow programs to switch between contexts.

Generally, programs acquire and release *external resources* through which they interact with the real world. These external resources are part of a larger context outside the program. By their very nature, they are stateful. In some scenarios, context switches should also back up and restore the state of these external resources.

Resource safety means that a program may only interact with external resources after it has acquired them, but before it has released them. Moreover, resources should be released eventually. The safe management of resources in presence of exceptions requires special care. For this reason, different mechanisms have been proposed and implemented, for example Resource Acquisition Is Initialization, `try-finally` blocks, or `defer` statements.

In a language with effect handlers, even more general non-local control flow has to be taken into account. Effect handlers capture the current continuation and potentially resume it later, or even multiple times. Mechanisms for finalization in presence of exceptions do not account for these. Abstractly, parts of the external context would have to be saved and later restored. Concretely, if resources have already been freed beforehand, resuming the program leads to undefined behavior, when already released external resources, such as file handles, are accessed.

In this paper, we propose a mechanism to improve the state of the art in handling of scarce resources in presence of general effect handlers. More specifically, in addition to the standard *return clause*, we propose to add two optional clauses to effect handlers: a *suspend clause* and a *resume clause*. They allow for user-defined actions to be executed upon unwinding respectively rewinding the control stack. These actions may release but also re-acquire resources when possible. When not possible, they allow for safely aborting the program in a controlled way. Together with the standard *return clause* these generalize the `try-finally` construct from exception handlers to effect handlers.

We formalize our ideas as the formal language System $\Xi_{\mathfrak{q}\rightarrow}$ equipped with a type system and operational semantics. Furthermore, we show the soundness of System $\Xi_{\mathfrak{q}\rightarrow}$ by proving progress and preservation. Finally, we state and prove a theorem of resource safety: even in presence of effect handlers with unrestricted continuation use, including from within finalizers, we guarantee that active resources are acquired and inactive ones are released. Our approach is complementary to existing work on statically ruling out undesired interaction between non-linear continuation use and external resources [Brachthäuser and Leijen 2023; Tang et al. 2024].

In summary, this paper makes the following contributions:

- **Calculus**: We present a formal calculus System $\Xi_{\mathfrak{q}\rightarrow}$ featuring lexical effect handlers endowed with resource finalization clauses.
- **Type System**: We present a type system for System $\Xi_{\mathfrak{q}\rightarrow}$.
- **Operational Semantics**: We equip System $\Xi_{\mathfrak{q}\rightarrow}$ with operational semantics by defining an abstract machine for evaluation.
- **Soundness**: Using the operational semantics and the type system, we prove the soundness, that is, progress and preservation for System $\Xi_{\mathfrak{q}\rightarrow}$.
- **Resource Safety**: We state and prove that acquiring and releasing of resources is well-bracket.
- **Implementation**: We implement the presented ideas as an extension of the research language Effekt [Brachthäuser et al. 2020].

This paper is structured into the following sections: In Section 2 we motivate the need for backtracking of heap resources and the safe management of external resources by presenting a basic parser combinator library for the Effekt language. Next, in Section 3 we present System $\Xi_{\leftrightarrow}$, that is, its syntax, type system and operational semantics. Additionally, we also prove the soundness of System $\Xi_{\leftrightarrow}$ and present the theorem of resource safety. In Section 4 we discuss the practical implementation of System $\Xi_{\leftrightarrow}$ in the Effekt language and highlight interesting aspects. We conclude this paper by presenting related work in Section 6 and summarizing our findings in Section 7.

2 Motivation

In this section, we explain basic usage of effect handlers and then motivate the need for dynamic wind. We start by building a parser combinator based on effect handlers [Brachthäuser et al. 2020; Leijen 2016]. We present these examples in Effekt, a language with lexical effect handlers, for which we have implemented our proposed finalization mechanism.

2.1 The Fail Effect: Modeling Exceptions

When programming with effect handlers, we separate programs that use effects from their context, which handles the effects. The two interact through a shared interface: the *effect signature*. Effectful programs use these effects with the keyword `do`. For example, we define a function that converts a character to an integer when it corresponds to a digit and fails otherwise.

```
effect fail(): Nothing
def digitToInt(char: Char): Int / fail =
  char match {
    case '0' ⇒ 0
    ...
    case _ ⇒ do fail()
  }
```

The type of `digitToInt` not only communicates the types of the parameter and return value but also that it *uses* the `fail` effect.

We handle effectful programs very much in the same way we handle exceptions. For example, we reify the potentially failing computation into an optional type.

```
def option[R] { program: () ⇒ R / fail }: Option[R] =
  try { Some(program()) }
  with fail { () ⇒ None() }
```

The higher-order function `option` accepts a program that might fail. When it succeeds, it returns the result of type `R` wrapped as `Some(result)`. When it fails, it returns `None()`. For example, `option { digitToInt('3') }` results in `Some(3)`.

2.2 The Read Effect: Resuming Computation

Effects generalize exceptions in that handlers have the ability to resume the program with the result of the effect operation. For example, we can define an effect `read` that abstracts over reading from character input streams. Using the `read` effect operation, we can define a function that parses digits. Of course, as before, the conversion to an integer might fail.

```
effect read(): Char
def digit(): Int / {read, fail} = digitToInt(do read())
```

A program can have a set of effects, here for example {read, fail}. They are inferred and accumulated from subprograms. We can handle the read effect in different ways to feed characters from different sources. For example from an array of characters.

```
def feed[R](input: Array[Char]) { reader: () ⇒ R / read }: R = {
  var position = 0
  try { reader() }
  with read { () ⇒
    val p = position
    if (p < input.size) { position = p + 1; resume(input.get(p)) }
    else { resume('\0') }
  }
}
```

In the handler feed, we keep track of the current position in the array using a local mutable variable position. When this position is within the array bounds, we resume with the character at this position. Otherwise, we resume with the null character, signaling that the end-of-file has been reached.

2.3 The Fork Effect: Resuming Multiple Times

Effect handlers have the ability to resume more than once, each time with a different value. For modelling non-deterministic computations, the effect fork can be used. Intuitively, it chooses between returning true in one world and false in another one. We can use it to choose the number of times an action should be performed.

```
effect fork(): Bool
def many { action: () ⇒ Unit / {} }: Unit / fork = while (do fork()) { action() }
```

While the type of the parameter action indicates no effects, many can still be called with actions that do have effects, thanks to contextual effect polymorphism [Brachthäuser et al. 2020]. We handle the fork effect with backtracking search.

```
def backtrack[R] { program: () ⇒ Option[R] / fork }: Option[R] =
  try { program() }
  with fork { () ⇒ resume(true) match {
      case None() ⇒ resume(false)
      case Some(result) ⇒ Some(result)
    }
  }
```

Here, we assume the program returns an optional result. To handle the fork effect, we first try to resume with true, and when we get no result, we resume with false.

2.4 The Parser Effect: Composing Effect Handlers

Having introduced all necessary ingredients, we can combine the effects into a parser effect by defining an alias for the set containing all three.

```
effect Parser = {read, fail, fork}
```

Using the parser effect, we can compose parsers similar to a parser combinator library [Hutton and Meijer 1998], but in direct style and freely combining it with imperative control flow, local mutable variables, and other effects — for example, for parsing a sequence of digits into a number.

```
def number(): Int / Parser = {
  var n = 0
  many { n = 10 * n + digit() }
  return n
}
```

The function `number` accumulates the result in a local mutable variable `n` by using the combinator `many` for non-deterministically choosing to continue iteration or to stop. Since `many` requires the context to handle the `fork` effect and `digit` the `read` and `fail` effect, `number` requires the context to handle the union of these effects. Of course, it is possible to combine these parsers into larger ones. For example, for parsing a pair of numbers separated by a non-digit character.

```
def numbers(): (Int, Int) / Parser = {
  val n₁ = number()
  val _ = do read()
  val n₂ = number()
  return (n₁, n₂)
}
```

Finally, we can handle the `Parser` effect by composing the three previously defined handlers.

```
def parse[R](input: Array[Char]) { parser: () ⇒ R / Parser }: Option[R] / {} =
  backtrack { option { feed(input) { parser() } } }
```

Given the input of `"112 21"`, running `parse(input) { numbers() }` results in the pair of integers `(112, 21)`.

2.5 Backtracking Global State

For the correct behavior of these parsers, it is of vital importance that the position in the array is backed up and restored upon the second resumption in handler `backtrack`. In Effekt, as `position` is a local mutable variable, this behavior follows naturally. If, however, we use a global mutable reference to keep track of the position, the parser exhibits wrong behavior.

Using a global mutable heap-allocated reference, running `parse(input) { numbers() }` with the same input `"112 21"` as before, would yield the pair of integers `(112, 1)`. As the reference is not stack-allocated, it is not automatically captured and restored by the stack unwind and rewind process process triggered by the `backtrack` handler and `fork` effect.

In this paper, we present dynamic wind for effect handlers. To this end, we add two new kinds of clauses: `on suspend` and `on resume`. Using these, we can keep the position in a global mutable reference, yet have the desired backtracking behavior.

```
def feedGlobal[R](input: Array[Char]) { reader: () ⇒ R / read }: R = {
  val position = ref(0) // a global mutable reference
  try { reader() }
  with read { () ⇒
    val p = position.get()
    if (p < input.size) { position.set(p + 1); resume(input.get(p)) }
    else { resume('\0') }
  } on suspend { position.get() }
  on resume { p ⇒ position.set(p) }
}
```

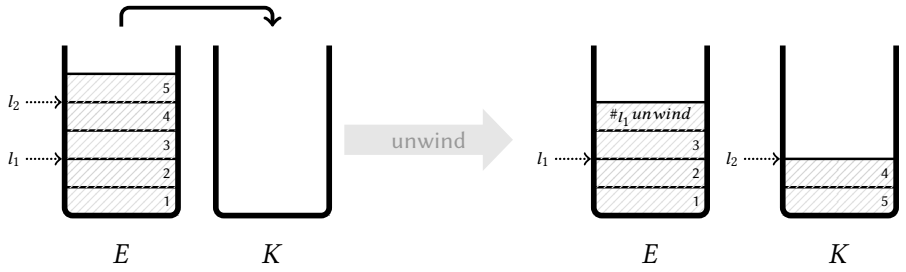


Fig. 1. During capturing the continuation, the unwinding process is paused to execute the suspend clause of each traversed handler. For remembering to later continue unwinding, a $\#_{l_1}$ **unwind** frame is pushed onto the captured continuation K before executing the suspend clause.

Intuitively, the statement in `on suspend` is executed whenever the computation is suspended, and the statement in `on resume` whenever it is resumed. Moreover, the result of executing the `on suspend` clause will be stored and passed to the `on resume` clause.

In much the same way, we can checkpoint more complex mutable data structures upon suspension and restore them upon resumption. For example, we can backtrack external resources, for example the position of a cursor in a file, or even close and reopen the file. This way, we can reuse our backtracking parser for directly parsing from a file, which we demonstrate in Section 4.1.

2.6 Resource Management

While one of the intended uses of these mechanisms is for backtracking external resources, our `on suspend` can serve a similar purpose as `finally` clauses found in Java and related languages. For example, we can close a file whenever the control flow leaves the enclosing block.

```
val file = open("example.txt", WriteOnly())
try { ... }
on suspend { close(file) }
on resume { _ ⇒ do throw("re-entering scope") }
on return { x ⇒ close(file); return x }
```

There are two ways the control flow can leave the block: when returning from the handler after computing the resulting value and when forwarding an effect operation to an outer handler. In both cases, we close the file using the corresponding clauses `on return` and `on suspend`. However, with effect handlers, we must also be prepared for the case when the control flow re-enters a block of code, potentially multiple times. In this case, here we have to throw an exception, because the file is already closed. We believe that in the absence of static checks for control-flow linearity [Brachthäuser and Leijen 2023; Tang et al. 2024; van Rooij and Krebbers 2025], this dynamic check is the best we can achieve. Unfortunately, this also means that even effect operations resuming exactly once cannot cross the handler protecting the file, as we illustrate in the next section.

2.7 Operational Intuition

Let us again consider the last program from the previous section. When calling an arbitrary effect operation in the body of `try`, the file is closed by the `suspend` clause. What happens if the corresponding handler resumes the continuation? The control flow returns to where the effect operation was called and continues. But the file is no longer open and the file handle thus invalid. Consequently, any subsequent actions using the file handle will fail.

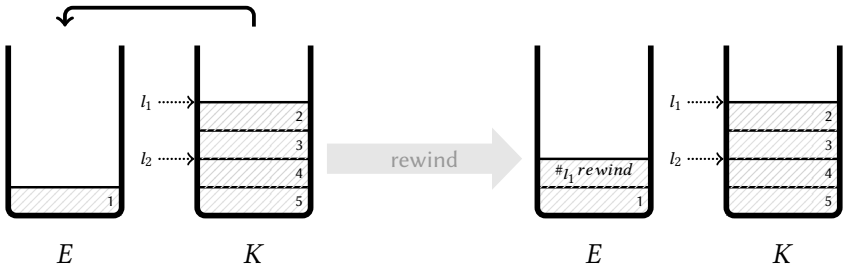


Fig. 2. Dual to the unwinding of the stack onto the captured continuation K, rewinding the continuation K onto the stack E prompts the execution of resume clauses. For remembering to continue rewinding the continuation onto the stack, a # **rewind** is pushed before executing the resume clause.

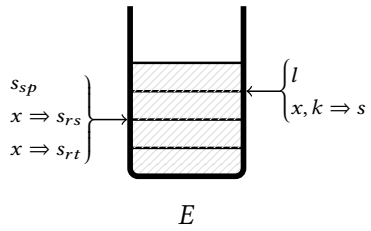


Fig. 3. A handler stack frame consists of a unique label l and the handler implementation itself. A finalizer frame consists of a suspend clause, a resume clause and a return clause.

It can be instructive to visualize the stack unwinding and continuation capturing, as can be seen in Figure 1. Since we cannot be sure whether the continuation will be discarded or resumed, we need to ensure that all resources are released *during* the continuation capturing (stack unwinding). Hence, we need to stop the unwinding process each time we encounter an effect handler that is not assigned the label we are searching for and finalize resources. In the case of Figure 1, we need to run the finalizer associated with the handler identified by the label l_2 and then remember to later continue the unwinding process in search of the label l_1. We also require that finalization works in the presence of multi-shot continuations and that the finalization clauses themselves may call effects and thereby capture the continuation.

While suspend clauses run during the continuation capture, we furthermore need re-initialization clauses for re-acquiring any released resources in case an effect operation resumes the continuation. We call these *resume clauses*. Dually, these clauses are run before a handler is rewound onto the stack, as shown in Figure 2. When used correctly, they allow library authors to guarantee that all resources are re-acquired before being accessed.

We also support *return clauses* with the semantics that whenever control flow exits a handler, the clause executed with the result of the computation in scope. Diverging from the semantics of languages like Koka [Leijen 2017] and Eff [Pretnar 2015], in our system it does not matter whether the handler returns normally from the handled program or via the handler itself; the return clause is executed in any case.

Importantly, when encountering a handler, not one handler frame, but two distinct frames are pushed onto the stack as visualized in Figure 3. First, a finalizer frame is pushed that consists of a suspend clause, a resume clause, and a return clause. Second, as is standard, a handler frame is pushed, consisting of a label l uniquely identifying it and the handler implementation.

When a finalizer frame is pushed onto the continuation during unwinding, the suspend clause is executed. Dually, when a finalizer frame is pushed back onto the stack during rewinding, the resume clause is executed. Lastly, if a finalizer frame is popped from the stack through a normal return (either from the handler or the handled program), the return clause is executed.

Given these new control flow constructs, as an alternative to throwing an exception, we can store the file in a local mutable reference, and re-open it when control flow re-enters the body.

```
var file = open("example.txt", WriteOnly())
try { ... }
on suspend { close(file) }
on resume { _ ⇒ file = open("example.txt", WriteOnly()) }
on return { x ⇒ close(file); return x }
```

Now, the `suspend` clause closes the file handle when the body yields, and the `resume` clause re-opens it when the continuation is resumed. Moreover, if the continuation is not resumed, the behavior is exactly, as desired as the file handle remains closed.

3 Formal Presentation

In this section, we present the syntax, type system, and operational semantics of System $\Xi_{\mathfrak{q}\mapsto}$, a language with effect handlers and dynamic wind. Furthermore, we state and prove the soundness of the language. Note that the presented calculus System $\Xi_{\mathfrak{q}\mapsto}$ is an extension to System Ξ as introduced by Brachthäuser et al. [2020] and relies on second-class functions to guarantee effect safety. We refer to these second-class functions as *blocks*.

3.1 Syntax

We start by presenting the syntax of System $\Xi_{\mathfrak{q}\mapsto}$ as shown in Figure 4. Besides the standard syntactic constructs of expressions and value literals, we also define so called blocks b. Blocks can either be a block variable f or a block literal of the form $\{\ (\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s\ \}$. This notation is to be understood as binding the names x_i and f_j within the scope of s.

Similarly, the statement $\mathbf{def}\ f\ =\ b;\ s$ binds the block b to the name f within the scope of s. We also have the familiar syntax for applying blocks $b(\overrightarrow{e_i},\ \overrightarrow{b_j})$. Furthermore, statements can be sequenced using the usual syntax $\mathbf{val}\ x\ =\ s;\ s$. Additionally, we support returning expressions where a statement is expected with $\mathbf{return}\ e$. Effect handlers $\mathbf{try}\ \{\ f \Rightarrow s\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$ are written in *explicit capability-passing style*, that is, the corresponding handler binds the block variable f (capability) in the handled statement s [Brachthäuser et al. 2020]. Compared to System Ξ, we add three new clauses to effect handlers: **on suspend**, **on resume**, and **on return**. We define **finally** as syntactic sugar for defining both an **on suspend** and an **on return** clause. We cover these new clauses in greater detail later. They perform dynamic wind and are at the focal point of enabling backtracking and finalization of external resources in the presence of effect handlers.

Finally, we define the syntax for the value Γ, block Δ and label Ξ environment. Values have value types τ, blocks have block types σ. The value environment Γ contains value bindings $x : \tau$ and the block environment contains Δ contains block bindings $f : \sigma$. The label environment Ξ contains a mapping from unique labels l to block types of capabilities. It is only used for proving the preservation of invariants in our abstract machine semantics.

3.2 Typing

The type system of System $\Xi_{\mathfrak{q}\mapsto}$ is defined in terms of three typing judgments for expressions, blocks, and statements (Figure 5).

Syntax:

Labels	l	::=	`@5ab` | `@42c` | ...	labels
Expressions	e	::=	x	expression variables
		|	v	values
Values	v	::=	() | 0 | 1 | ...	primitives
Blocks	b	::=	f	block variables
		|	$\{\ (\overrightarrow{x : \tau}, \overrightarrow{f : \sigma}) \Rightarrow s\ \}$	block implementation
Statements	s	::=	**def** $f\ =\ b;\ s$	block definition
		|	$b(\overrightarrow{e},\ \overrightarrow{b})$	block application
		|	**val** $x\ =\ s;\ s$	sequencing
		|	**return** e	returning
		|	**try** $\{\ f \Rightarrow s\ \}$ **with** h	effect handler
Handler	h	::=	$\{(\overrightarrow{x},\ k) \Rightarrow s\ \}$	handler implementation
			on suspend $\{\ s\ \}$	suspend clause
			on resume $\{\ x \Rightarrow s\ \}$	resume clause
			on return $\{\ x \Rightarrow s\ \}$	return clause

Definitions:

try $\{\ s_1\ \}$ **finally** $\{\ s_2\ \}$ $\stackrel{\text{def}}{=}$ **try** $\{\ s_1\ \}$ **on suspend** $\{\ s_2\ \}$ **on return** $\{\ x \Rightarrow s_2;$ **return** $x\ \}$

Types:

Value Types	τ	::=	Int | Bool | Unit | ...	base types
Block Types	σ	::=	$(\overrightarrow{\tau},\ \overrightarrow{\sigma}) \to \tau$	function types

Environments:

Value Environment	Γ	::=	$\emptyset$	empty environment
		|	$\Gamma,\ x : \tau$	value bindings
Block Environment	Δ		$\emptyset$	empty environment
		|	$\Delta,\ f : \sigma$	block bindings
Label Environment	Ξ		$\emptyset$	empty environment
		|	$\Delta,\ l : \overrightarrow{\tau} \to \tau$	block bindings

Fig. 4. Syntax System $\Xi_{\not\to}$

As noted earlier, there is a distinction being made between expressions and blocks, that is, blocks are treated as second-class [Levy 2001; Osvald et al. 2016], while expressions are first-class: blocks cannot be returned or stored. This is to ensure effect safety as blocks may close over capabilities and returning them would lead to capabilities (passed as blocks) escaping their respective scope. Though, Brachthäuser et al. [2022] show that this is not an inevitable necessity by presenting System C in which the captured capabilities of second-class blocks can be reified into the type of a first-class function, thereby ensuring they are still in scope upon usage. This distinction between potentially effectful, second-class computations (statements) and pure, first-class expressions is also reflected by two distinct type environments: Γ for mapping from value variables x to value types τ and Δ for mapping from block variables f to block types σ. Besides the aforementioned separation of values and blocks, the typing rules for blocks (Block), (Block-Var), and expressions (Lit), (Var) are standard.

Statements. The typing rules for sequencing, returning as well as applying and defining blocks are also completely standard. More interestingly, we recall the original typing rule for lexical effect handlers of System Ξ [Brachthäuser et al. 2020].

Block Typing. $\boxed{\Gamma \mid \Delta \mid \Xi \vdash b : \sigma}$

$$\frac{f : \sigma \in \Delta}{\Gamma \mid \Delta \mid \Xi \vdash f : \sigma}\ [\textsc{Block-Var}]$$

$$\frac{\Gamma, \overrightarrow{x_i : \tau_i} \mid \Delta, \overrightarrow{f_j : \sigma_j} \mid \Xi \vdash s : \tau}{\Gamma \mid \Delta \mid \Xi \vdash \{ (\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s \} : (\overrightarrow{\tau_i}, \overrightarrow{\sigma_j}) \to \tau}\ [\textsc{Block}]$$

Expression Typing. $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}\ [\textsc{Lit}] \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ [\textsc{Var}]$$

Statement Typing. $\boxed{\Gamma \mid \Delta \mid \Xi \vdash s : \tau}$

$$\frac{\Gamma \mid \Delta \mid \Xi \vdash s_1 : \tau_1 \qquad \Gamma, x : \tau_1 \mid \Delta \mid \Xi \vdash s_2 : \tau_2}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{val}\ x = s_1; s_2 : \tau_2}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{return}\ e : \tau}\ [\textsc{Ret}]$$

$$\frac{\overrightarrow{\Gamma \vdash e_i : \tau_i} \qquad \overrightarrow{\Gamma \mid \Delta \mid \Xi \vdash b_j : \sigma_j} \qquad \Gamma \mid \Delta \mid \Xi \vdash b : (\overrightarrow{\tau_i}, \overrightarrow{\sigma_j}) \to \tau}{\Gamma \mid \Delta \mid \Xi \vdash b(\overrightarrow{e_i}, \overrightarrow{b_j}) : \tau}\ [\textsc{App}]$$

$$\frac{\Gamma \mid \Delta \mid \Xi \vdash b : \sigma \qquad \Gamma \mid \Delta, f : \sigma \mid \Xi \vdash s : \tau}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{def}\ f = b; s : \tau}\ [\textsc{Def}]$$

$$\frac{\Gamma \mid \Delta, f : \overrightarrow{\tau_i} \to \tau_0 \mid \Xi, l : \overrightarrow{\tau_i} \to \tau_0 \vdash s : \tau \qquad \Gamma \mid \Delta \mid \Xi \vdash h : \overrightarrow{\tau_i} \to \tau_0 \mid \tau \Rightarrow \tau_3}{\Gamma \mid \Delta \mid \Xi \vdash \mathbf{try}\ \{ f \Rightarrow s \}\ \mathbf{with}\ h : \tau_3}\ [\textsc{Try-Finalize}]$$

Handler Typing. $\boxed{\Gamma \mid \Delta \mid \Xi \vdash h : \tau \to \tau \mid \tau \Rightarrow \tau}$

$$\frac{\begin{array}{l}\Gamma, \overrightarrow{x_i : \tau_i} \mid \Delta, k : \tau_0 \to \tau \mid \Xi \vdash s : \tau \qquad \Gamma \mid \Delta \mid \Xi \vdash s_1 : \tau_1 \\ \Gamma, x_1 : \tau_1 \mid \Delta \mid \Xi \vdash s_2 : \mathsf{Unit} \qquad \Gamma, x : \tau \mid \Delta \mid \Xi \vdash s_3 : \tau_3\end{array}}{\begin{array}{lll}\Gamma \mid \Delta \mid \Xi \vdash & \{ (\overrightarrow{x_i}, k) \Rightarrow s \} & : \overrightarrow{\tau_i} \to \tau_0 \mid \tau \Rightarrow \tau_3 \\ & \mathbf{on\ suspend}\ \{ s_1 \} & \\ & \mathbf{on\ resume}\ \{ x_1 \Rightarrow s_2 \} & \\ & \mathbf{on\ return}\ \{ x \Rightarrow s_3 \} &\end{array}}\ [\textsc{Try-Handler}]$$

Fig. 5. Typing of System $\Xi_{\leftrightarrow}$.

$$\frac{\Gamma \mid \Delta, f : \overrightarrow{\tau_i} \to \tau' \mid \Xi \vdash s_1 : \tau \qquad \Gamma, \overrightarrow{x_i : \tau_i} \mid \Delta, k : \tau' \to \tau \mid \Xi \vdash s_2 : \tau}{\Gamma \mid \Delta \mid \Xi \vdash \textbf{try}\ \{ f \Rightarrow s_1 \}\ \textbf{with}\ \{ (\overrightarrow{x_i}, k) \Rightarrow s_2 \} : \tau}$$

The handler introduces a capability, which is bound to f with type $\overrightarrow{\tau_i} \to \tau'$, within the scope of s_1. This capability's implementation is given by s_2 which may use the capability's parameters $\overrightarrow{x_i}$ and the captured continuation k. We notice that the left-hand side of the block type of f coincides with the type of the parameters $\overrightarrow{x_i}$. Additionally, the continuation k expects an argument of type τ' as this is exactly the return type the capability of f promises to its caller. Since the handler's implementation may choose to not resume, that is, not call the continuation k, the types of s_1 and s_2 have to match such that the resulting type of the statement is τ.

$$\frac{\begin{array}{c}\Gamma \mid \Delta \mid \Xi \vdash s : \tau \qquad \Gamma \mid \Delta \mid \Xi \vdash s_1 : \tau_1 \\ \Gamma, x_1 : \tau_1 \mid \Delta \mid \Xi \vdash s_2 : \mathsf{Unit} \qquad \Gamma, x : \tau \mid \Delta \mid \Xi \vdash s_3 : \tau_3\end{array}}{\Gamma \vdash \textbf{try}\ \{ s \}\ \textbf{on suspend}\ \{ s_1 \}\ \textbf{on resume}\ \{ x_1 \Rightarrow s_2 \}\ \textbf{on return}\ \{ x \Rightarrow s_3 \} : \tau_3}$$

We extend handler statements with three clauses to enable dynamic wind for effect handlers. Compared to the previous typing rule, we can see that the overall type of the statement is determined by the type of s_3. Furthermore, $x : \tau$ is in scope for s_3, that is, the **return** clause accepts the return value of s as its parameter. Apart from returning, it is also possible for the control flow to exit the current effect handler by calling capabilities introduced by outer handlers. The **suspend** clause consists of s_1, while the **resume** clause takes its return value as a parameter $x_1 : \tau_1$. Note that these clauses do not alter the overall type of the handler.

Combining the previous two typing rules results in the rules (Try-Finalize) and (Try-Handler), which are at the core of the typing rules of System $\Xi_{\mathfrak{q}\to}$ in Figure 5. (Try-Finalize) checks a **try** statement against the type τ_3. Additionally, the body s is checked against τ with the capability $f : \overrightarrow{\tau_i} \to \tau_0$ in scope. For typing the handler h and the finalizer clauses therein, (Try-Handler) is invoked. (Try-Handler) type-checks the handler h against a capability type $\overrightarrow{\tau_i} \to \tau_0$ and a return clause type $\tau \Rightarrow \tau_3$. When checking the handler body s against τ, the capability's parameters $\overrightarrow{x_i : \tau_i}$ and the continuation's type $\tau_0 \to \tau$ are brought into scope. Also, note that the suspend clause must yield Unit while being passed the return value $x_1 : \tau_1$ of the suspend clause. Finally, the return clause is being passed the result of the try body or that of the handler's implementation and transforms the intermediate result of type τ it into a value of type τ_3. Notably, the continuation's type is not altered by the return clause.

The remaining typing rules for statements Val, Ret, App and Def are completely standard except for the separation of expressions and blocks in Def, where f is added not to Γ but Δ.

3.3 Semantics

We now turn to presenting the operational semantics of System $\Xi_{\mathfrak{q}\to}$. First, we introduce the syntax of an abstract machine. Then, we define the reduction rules for the abstract machine. Lastly, we state and prove theorems about machine reduction. To do so, we extend typing to machine states, which we omit from this presentation.

3.3.1 Abstract Machine. The syntax defined in Figure 6 is to be seen as an extension of the syntax of System $\Xi_{\mathfrak{q}\to}$ given in Figure 4. The new runtime constructs are denoted by a leading #.

We introduce two new kinds of block values. First, capability calls of the form $\#_l$ **cap** that are introduced by handlers and are annotated with a label l. Second, resumptions of the form # **resume**$(l, \{(\overrightarrow{x}, k) \Rightarrow s\}, K)$, taking a label l, a handler implementation $\{(\overrightarrow{x}, k) \Rightarrow s\}$ and continuation K as arguments.

Syntax for the Operational Semantics and Abstract Machine:

Blocks	b	::=	f	block variables
		|	w	block values
		|	$\#_l$ **cap**	capabilities
		|	$\#$ **resume**$(l,\ \{(\overrightarrow{x},\ k) \Rightarrow s\},\ K)$	resumptions
Block values	w	::=	$\{\ (\overrightarrow{x : \tau}, \overrightarrow{f : \sigma}) \Rightarrow s\ \}$	block implementation
Runtime Stack	E	::=	$\bullet$	
		|	$\#_l\ \{\ \square\ \}$ **with** $\{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E$	handler
		|	$\mathcal{F}\ ::\ E$	finalizer
		|	$F\ ::\ E$	frame
Continuation	K	::=	$\bullet$	
		|	$\#_l\ \{\ \square\ \}$ **with** $\{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E$	handler
		|	$(\mathcal{F},\ v)\ ::\ K$	finalizer, value pair
		|	$F\ ::\ K$	
Frames	F	::=	**val** $x\ =\ \square;\ s$	sequencing
		|	$\#_l$ **unwind**$(\overrightarrow{v},\ \mathcal{F},\ K)$	unwinding
		|	$\#$ **rewind**$(v,\ \mathcal{F},\ K)$	rewinding
Finalizer	$\mathcal{F}$	::=	$\{\ \square\ \}$ **on suspend** $\{\ s\ \}$ **on resume** $\{\ x \Rightarrow s\ \}$ **on return** $\{\ x \Rightarrow s\ \}$	
Abstract Machine	M	::=	$\langle\ s \mid E\ \rangle$	reduction mode
		|	$\langle\ s \mid E \looparrowright K\ \rangle$	unwind mode
		|	$\langle\ s \mid E \hookleftarrow K\ \rangle$	rewind mode

Fig. 6. Syntax of the abstract machine of System $\Xi_{\looparrowright}$

Next, we define the syntax of the runtime stack and continuation. The runtime stack is either empty ($\bullet$), contains stack frames F, or a runtime handler $\#_l\ \{\ \square\ \}$ **with** $\{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}$ annotated with a label l. Importantly, the stack may also contain finalizer frames $\mathcal{F}$, consisting of a **suspend** clause, a **resume** clause and a **return** clause. Compared to the syntax of System $\Xi_{\looparrowright}$, we separate handlers from finalizers. The syntax of the runtime continuation almost mirrors that of the runtime stack, the only difference being the different syntax for finalizers $\mathcal{F}$: finalizer frames on the continuation are finalizer-value pairs.

The syntax of frames F is shared by both the runtime continuation and stack. A frame F is either **val** $x\ =\ \square;\ s$ for sequencing, an unwind frame $\#_l$ **unwind**$(\overrightarrow{v},\ \mathcal{F},\ K)$ annotated with a label l and containing multiple values, a label, a finalizer frame $\mathcal{F}$ and a continuation as arguments, or a rewind frame $\#$ **rewind**$(v,\ \mathcal{F},\ K)$ consisting of a value v, finalizer $\mathcal{F}$ and continuation K.

Lastly, the syntax for the abstract machine itself consists of three cases. Each case corresponds to one of three modes the machine may be in. In reduction mode, the machine is of the form $\langle\ s \mid E\ \rangle$, in unwind mode $\langle\ s \mid E \looparrowright K\ \rangle$ and in rewind mode $\langle\ s \mid E \hookleftarrow K\ \rangle$.

3.3.2 Evaluation. Figure 7 presents the operational semantics in terms of an abstract machine. There are three states the machine can be in.

Reduction Mode $\langle\ s \mid E\ \rangle$. States of the form $\langle\ s \mid E\ \rangle$ are used to perform standard machine reductions. For brevity, we omit the trivial congruence rules for expressions. The rules *(def)* and *(app)* reduce function definition and block applications in the expected way without a context. The rules *(cong)*, *(pop)* and *(push)* perform standard reductions of pushing and popping frames of the stack. The rule *(return)* reduces **return** v in the context of a handler frame by simply popping the handler frame from the stack.

The rule *(return')* reduces the return of a value v within the context of a finalizer frame to $\mathcal{F}.\mathbf{return}[x \mapsto v]$, where $\mathcal{F}.\mathbf{return}$ is the body of the **return** clause. Recall from Figure 6 that $\mathcal{F}$ stands for a finalizer frame consisting of a suspend, resume and return clause. The **return** clause's body receives a value v of as argument x and continues the evaluation in reduction mode.

The rule *(try)* introduces both a handler frame $\#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}$ and a finalizer frame $\mathcal{F}(h)$ that is extracted from the handler h in the obvious way. Furthermore, a capability $\#_l\ \mathbf{cap}$ is substituted for f in s with a fresh label l coinciding with the label given to the introduced handler frame.

Unwind Mode $\langle\ s \mid E \looparrowright K\ \rangle$. We model the process of unwinding the stack E in search for a certain delimiter (label) l, thereby capturing the continuation K. E is the remainder of the stack the search continues on. By construction, this state is always of the form $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright K\ \rangle$.

The rule *(cap)* engages the unwind mode with the empty continuation $\bullet$ when a capability call occurs. During the unwind process, stack frames F are simply pushed from the stack E onto the captured continuation K as expressed by the stepping rule *(unwind)*. The rule *(unwind')* serves a similar purpose and just pushes a handler from the stack onto the continuation.

Eventually, we will find the appropriate handler with the correct label l. Then, *(handle)* disengages the unwind mode. Consequently, the handler's implementation s is evaluated in reduction mode with the capability's arguments $\overrightarrow{v_i}$ substituted for $\overrightarrow{x_i}$ and a resume block of form $\#\ \mathbf{resume}(l,\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)$ for k.

If a finalizer frame is found while unwinding the stack, *(suspend)* prompts the evaluation of the finalizer's suspend clause, denoted by $\mathcal{F}.\mathbf{suspend}$. Since we originally wanted to search for the delimiter l and thus have to remember to unwind the stack E later on, we save all the needed information to reinstate the encountered finalizer frame later in an unwind frame of the form $\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)$ and push it onto the stack E. Notice that the machine now enters reduction mode and temporarily suspends the unwinding process.

When encountering a $\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)$ frame, *(suspend')* reinstantiates the capability call $\#_l\ \mathbf{cap}(\overrightarrow{v})$ and re-engages unwind mode. Importantly, the finalizer frame is now pushed back onto the continuation, together with the result v of evaluating the suspend clause for later passing it to the resume clause when potentially rewinding.

Rewind Mode $\langle\ s \mid E \hookleftarrow K\ \rangle$. In this mode, we rewind the captured continuation K back onto the stack E until K is of form $\bullet$. By construction, the machine is always of form $\langle\ \mathbf{return}\ v \mid E \hookleftarrow K\ \rangle$ in rewind mode.

The rule *(cont)* starts the transition from reduction mode to rewind mode if a resume block $\#\ \mathbf{resume}(l,\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)$ is applied to a value v in reduction mode. Using the label l, the handler h, and the captured continuation K stored in the resume block, we reinstate the handler back onto the stack E and start the process of rewinding the captured continuation. Since applying a value v to a resume block can be seen as calling the continuation, we need to return v back to the call-site of the capability that caused the unwind process.

Symmetric to *(unwind)*, *(rewind)* moves frames F from the continuation K back onto E and *(rewind')* does the same for handler frames. Similarly to how *(suspend)* defines the behavior when encountering finalizers during the unwind process, *(resume)* defines the behavior of encountering finalizers during the *rewind* process. When pushing a handler from the captured continuation K back onto the stack E, we have to evaluate the resume clause $\mathcal{F}.\mathbf{resume}$ and substitute the result of the previously evaluated suspend clause v' for x. Furthermore, we need to remember to continue rewinding the stack later on. Similarly to the unwind frame $\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)$, the rewind frame $\#\ \mathbf{rewind}(v',\ \mathcal{F},\ K)$ fulfills the same role for the rewind process.

Dual to *(suspend')*, *(resume')* unpacks the previously encountered finalizer frame from a rewind frame, pushes it onto the stack E and re-engages the rewind mode using K and v'.

Lastly, *(stop)* terminates the rewind mode if the captured continuation has been rewound completely onto the stack E.

3.3.3 Evaluation Context. The decision to execute the suspend, resume, and return clause in the outer context, where the finalizer frame is not yet pushed back onto the stack, may seem arbitrary but there are good reasons for it. Running the suspend clause in the inner context (with the finalizer frame pushed back onto the stack) is problematic if it uses capabilities since this would again prompt the unwinding of the stack. If the stack is unwound while executing the suspend clause, the same clause would be triggered again, causing an infinite loop. The same argument can be made for the return clause.

If the resume clause were run in the inner context and uses capabilities, the suspend clause would be triggered. For example, if the suspend clause closes a file handle and the resume clause opens it again, this would immediately close the file handle again before resuming the continuation, leading to accesses to invalid file handles. It furthermore violates the intuitive requirement that the suspend clause and resume clause should cancel out one another. This is only the case if for each resume clause execution, the suspend clause is executed exactly once.

3.3.4 Examples. In the following, we present two examples to convey a better understanding of the operational semantics. As the reduction can be quite involved, even for small examples, we color code the three different parts of the abstract machine. The statement under reduction is highlighted in *yellow*, the runtime stack in *blue* and the captured continuation in *red*.

Example 3.1. Consider the following example, where we reduce the simple program

$$\textbf{try}\ \{\ f \Rightarrow \textbf{val}\ x\ =\ f(1);\ \textbf{return}\ x\ +\ 1\ \}\ \textbf{with}\ h$$

where

$$h\ :=\ \{\ (x,\ k) \Rightarrow k(x\ *\ 2)\ \}.$$

Evaluating the program using the operational semantics yields the following reduction steps:

$$\begin{array}{ll}
\langle\ \textbf{try}\ \{\ f \Rightarrow \textbf{val}\ x\ =\ f(1);\ \textbf{return}\ x\ +\ 1\ \}\ \textbf{with}\ h\ \mid\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{val}\ x\ =\ \#_{@1}\ \textbf{cap}(1);\ \textbf{return}\ x\ +\ 1\ \mid\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \#_{@1}\ \textbf{cap}(1)\ \mid\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \#_{@1}\ \textbf{cap}(1)\ \mid\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \looparrowright\ \bullet\ \rangle & \longrightarrow \\
\langle\ \#_{@1}\ \textbf{cap}(1)\ \mid\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \looparrowright\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \#\ \textbf{resume}(@1,\ h,\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \bullet\ )(1\ *\ 2)\ \mid\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{return}\ 2\ \mid\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \hookleftarrow\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{return}\ 2\ \mid\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \hookleftarrow\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{return}\ 2\ \mid\ \textbf{val}\ x\ =\ \square;\ \textbf{return}\ x\ +\ 1\ ::\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{return}\ 3\ \mid\ \#_{@1}\ \{\ \square\ \}\ \textbf{with}\ h\ ::\ \bullet\ \rangle & \longrightarrow \\
\langle\ \textbf{return}\ 3\ \mid\ \bullet\ \rangle &
\end{array}$$

Example 3.2. Next, consider the following, more complex program that uses **suspend** and **resume** clauses. Within those, we use a capability introduced at an outer handler.

$$\textbf{try}\ \{\ f \Rightarrow \textbf{try}\ \{\ g \Rightarrow f(v_0)\ \}\ \textbf{with}\ h_2\ \}\ \textbf{with}\ h_1$$

where

$$\begin{aligned}
h_1\ &:=\ \{\ (x,\ k) \Rightarrow k(v_1)\ \} \\
h_2\ &:=\ \textbf{on suspend}\ \{\ f(v_2)\ \}\ \textbf{on resume}\ \{\ x \Rightarrow f(v_3)\ \}
\end{aligned}$$

Reduction without context:

(def)	$\mathbf{def}\ f\ =\ w;\ s$	$\longrightarrow$	$s[f \mapsto w]$
(app)	$(\{\ (\overrightarrow{x_i},\ \overrightarrow{f_j}) \Rightarrow s\ \})(\overrightarrow{v_i},\ \overrightarrow{w_j})$	$\longrightarrow$	$s[\overrightarrow{x_i \mapsto v_i},\ \overrightarrow{f_j \mapsto w_j}]$

Machine reductions:

Standard Machine Reductions

(cong)	$\langle\ s \mid E\ \rangle$	$\longrightarrow$	$\langle\ s' \mid E\ \rangle$ if $s \longrightarrow s'$
(pop)	$\langle\ \mathbf{return}\ v \mid \mathbf{val}\ x\ =\ \square;\ s\ ::\ E\ \rangle$	$\longrightarrow$	$\langle\ s[x \mapsto v] \mid E\ \rangle$
(push)	$\langle\ \mathbf{val}\ x\ =\ s_1;\ s_2 \mid E\ \rangle$	$\longrightarrow$	$\langle\ s_1 \mid \mathbf{val}\ x\ =\ \square;\ s_2\ ::\ E\ \rangle$
(return)	$\langle\ \mathbf{return}\ v \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E\ \rangle$	$\longrightarrow$	$\langle\ \mathbf{return}\ v \mid E\ \rangle$
(return')	$\langle\ \mathbf{return}\ v \mid \{\ \square\ \}\ \mathcal{F}\ ::\ E\ \rangle$	$\longrightarrow$	$\langle\ \mathbf{return}(\mathcal{F})[x \mapsto v] \mid E\ \rangle$

Installing Effect Handlers

(try) $\langle\ \mathbf{try}\ \{\ f \Rightarrow s\ \}\ \mathbf{with}\ h \mid E\ \rangle \longrightarrow$
$\langle\ s[f \mapsto \#_l\ \mathbf{cap}] \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ \mathcal{F}(h)\ ::\ E\ \rangle$ where l fresh

Finalization

(cap) $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E\ \rangle \longrightarrow$
$\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright \bullet\ \rangle$

(unwind) $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid F\ ::\ E \looparrowright K\ \rangle \longrightarrow$
$\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright F\ ::\ K\ \rangle$

(unwind') $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E \looparrowright K\ \rangle \longrightarrow$
$\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright \#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ K\ \rangle$
where $l \neq l'$

(suspend) $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \mathcal{F}\ ::\ E \looparrowright K\ \rangle \longrightarrow$
$\langle\ \mathcal{F}.\mathbf{suspend} \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)\ ::\ E\ \rangle$

(suspend') $\langle\ \mathbf{return}\ v \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)\ ::\ E\ \rangle \longrightarrow$
$\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright (\mathcal{F},\ v)\ ::\ K\ \rangle$

(resume) $\langle\ \mathbf{return}\ v \mid E \hookleftarrow (\mathcal{F},\ v')\ ::\ K\ \rangle \longrightarrow$
$\langle\ \mathcal{F}.\mathbf{resume}[x \mapsto v'] \mid \#\ \mathbf{rewind}(v,\ \mathcal{F},\ K)\ ::\ E\ \rangle$

(resume') $\langle\ \mathbf{return}\ v \mid \#\ \mathbf{rewind}(v',\ \mathcal{F},\ K)\ ::\ E\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v' \mid \mathcal{F}\ ::\ E \hookleftarrow K\ \rangle$

(rewind) $\langle\ \mathbf{return}\ v \mid E \hookleftarrow F\ ::\ K\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v \mid F\ ::\ E \hookleftarrow K\ \rangle$

(rewind') $\langle\ \mathbf{return}\ v \mid E \hookleftarrow \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ K\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E \hookleftarrow K\ \rangle$

(handle) $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E \looparrowright K\ \rangle \longrightarrow$
$\langle\ s[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \#\ \mathbf{resume}(l,\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)] \mid E\ \rangle$
where $l = l'$

(cont) $\langle\ \#\ \mathbf{resume}(l,\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)(v) \mid E\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E \hookleftarrow K\ \rangle$

(stop) $\langle\ \mathbf{return}\ v \mid E \hookleftarrow \bullet\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v \mid E\ \rangle$

Fig. 7. Abstract machine semantics of System $\Xi_{\looparrowright}$.

For brevity, we only show the most interesting reduction steps, omit the rest and only push a finalizer frame for the inner handler instead of a handler and finalizer frame.

$$
\begin{array}{lll}
\langle\ \mathbf{try}\ \{\, f \Rightarrow \mathbf{try}\ \{\, g \Rightarrow f(v_0)\,\}\ \mathbf{with}\ h_2\,\}\ \mathbf{with}\ h_1 \mid \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \#_{@1}\ \mathbf{cap}(v_0) \mid \{\,\square\,\}\ h_2 :: \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet \curvearrowright \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \#_{@1}\ \mathbf{cap}(v_2) \mid \#_{@1}\ \mathbf{unwind}(v_0,\ h_2,\ \bullet) :: \#_{@1}\ \{\,\bullet\,\}\ \mathbf{with}\ h_1 :: \bullet\ \rangle\ {}^{(1)} & \longrightarrow^* & \\
\langle\ \#_{@1}\ \mathbf{cap}(v_2) \mid \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet \curvearrowright \#_{@1}\ \mathbf{unwind}(v_0,\ h_2,\ \bullet) :: \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \#\,\mathbf{resume}(@1,\ h_1,\ \#_{@1}\ \mathbf{unwind}(v_0,\ h_2,\ \bullet) :: \bullet)(v_1) \mid \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \mathbf{return}\ v_1 \mid \#_{@1}\ \mathbf{unwind}(v_0,\ h_2,\ \bullet) :: \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet\ \rangle\ {}^{(2)} & \longrightarrow & \\
\langle\ \#_{@1}\ \mathbf{cap}(v_0) \mid \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet \curvearrowright (\{\,\square\,\}\ h_2,\ v_1) :: \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \mathbf{return}\ v_1 \mid \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet \curvearrowleft (\{\,\square\,\}\ h_2,\ v_1) :: \bullet\ \rangle\ {}^{(3)} & \longrightarrow & \\
\langle\ \#_{@1}\ \mathbf{cap}(v_3) \mid \#\,\mathbf{rewind}(v_1,\ h_2,\ \bullet) :: \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet\ \rangle\ {}^{(4)} & \longrightarrow^* & \\
\langle\ \mathbf{return}\ v_1 \mid \#\,\mathbf{rewind}(v_1,\ h_2,\ \bullet) :: \#_{@1}\ \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \mathbf{return}\ v_1 \mid \{\,\square\,\}\ h_2 :: \{\,\square\,\}\ \mathbf{with}\ h_1 :: \bullet \curvearrowleft \bullet\ \rangle & \longrightarrow^* & \\
\langle\ \mathbf{return}\ v_1 \mid \bullet\ \rangle &&
\end{array}
$$

This more complex example showcases that even when the **suspend** and **resume** clauses themselves are effectful, no stack unwinding or rewinding is forgotten.

Step *(1)* is interesting, because when unwinding the stack to find the handler with label $\#_{@1}$, we need to traverse a finalizer frame. Thus, we first need to execute the **suspend** clause (which leads to another effect $f(v_2)$), before we later continue unwinding the stack. We can see that the # **unwind** frame serves as a reminder for unwinding of the stack and also for discharging the finalizer frame such that it is not executed again when encountering effects in the **suspend** clause.

At step *(2)*, we have returned from the effect in the **suspend** clause and push the finalizer frame onto the continuation, together with the returned value, later to be used by the **resume** clause.

At step *(3)*, we traverse the finalizer frame again while rewinding the stack, thus, the **resume** clause is executed. However, we still have to remember to continue rewinding later, hence, a # **rewind** primitive is pushed onto the stack and again deactivates the finalizer frame such that the suspend clause is not triggered again by effects in the **resume** clause.

At step *(4)*, we then continue rewinding with the original return value of $f(v_0)$, yielding it as the final result.

3.4 Properties

In this section, we prove the theorems of progress (Theorem 3.3) and preservation (Theorem 3.4) with respect to the type system and operational semantics. Combining progress and preservation gives us soundness (Theorem 3.5) as usual.

3.4.1 *Progress.*

Theorem 3.3 (Progress). *If* $\vdash_m M : \tau$, *then there either exists a value* v *such that* $M = \langle\ \mathbf{return}\ v \mid \bullet\ \rangle$ *or a machine* M' *with* $M \longrightarrow M'$.

Proof. The proof is mostly straightforward. First, a case distinction on the abstract machine typing $\vdash_m M : \tau$ is made, followed by inversion and a case distinction on the typing derivation of *s*. □

3.4.2 *Preservation.*

Theorem 3.4 (Preservation). *If* $\vdash_m M : \tau$ *and there exists* $M \longrightarrow M'$, *then* $\vdash_m M' : \tau$.

Proof. The proof is by case distinction on the step $M \longrightarrow M'$ and continued application of inversion within each case. □

Machine reductions:

…

(return') $\langle\ \mathbf{return}\ v \mid \#_l\,\mathcal{F} :: E \mid R\ \rangle \longrightarrow$
$\langle\ \mathbf{return}(\mathcal{F})[x \mapsto v] \mid E \mid R(l) = 0\ \rangle$

(try) $\langle\ \mathbf{try}\ \{ f \Rightarrow s \}\ \mathbf{with}\ h \mid E \mid R\ \rangle \longrightarrow$
$\langle\ s[f \mapsto \#_l\ \mathbf{cap}] \mid \#_l\ \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x},\ k) \Rightarrow s_h \} :: \#_l\,\mathcal{F}(h) :: E \mid R(l) = 1\ \rangle$
where l fresh

(suspend) $\langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \#_{l'}\,\mathcal{F} :: E \leadsto K \mid R\ \rangle \longrightarrow$
$\langle\ s \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \#_{l'}\,\mathcal{F},\ K) :: E \mid R(l') = 0\ \rangle$

(resume') $\langle\ \mathbf{return}\ v \mid \#\ \mathbf{rewind}(v',\ \#_l\,\mathcal{F},\ K) :: E \mid R\ \rangle \longrightarrow$
$\langle\ \mathbf{return}\ v' \mid \#_l\,\mathcal{F} :: E \leftsquigarrow K \mid R(l) = 1\ \rangle$

Fig. 8. Instrumented abstract machine semantics of System $\Xi_{\leadsto}$. Changes compared to the previous machine semantics are highlighted in *gray*.

3.4.3 *Soundness.*

Theorem 3.5 (Soundness). *If* $\vdash_m M : \tau$, *there exists a step* $M \longrightarrow^* M'$ *and there exists no further step* $M' \longrightarrow M''$, *then* $\vdash_m M' : \tau$ *and there exists some value* v *such that* $M' = \langle\ \mathbf{return}\ v \mid \bullet\ \rangle$.

Proof. Directly follows from the continued, interleaved application of the theorems of Progress (Theorem 3.3) and Preservation (Theorem 3.4). □

3.4.4 Resource Safety. One of the most obvious and desirable properties is that all active resources are actually open and all others are closed. However, the previously presented operational semantics do not directly allow us to reason about the state of resources. Thus, in this section, we instrument the abstract machine in Figure 8 such that it is augmented to carry an additional resource environment R that maps abstract resource labels l to either an open state (1) or a closed state (0).

Commonly, **try** – **finally** statements are used in the following way:

```
val file = open(path)
try { ... read(file) ... }
finally { ... close(file) ... }
```

Before the **try** – **finally** statement is entered, a file handle is opened which then is subsequently read in the body and finalized by closing it again.

We capture this notion of initializing the resource before entering the body in the augmented rule of *(try)*. When pushing a finalizer frame onto the stack in step *(try)*, we also open the resource that is conceptually associated with this handler by setting $R(l) = 1$. Dually, in *(return')*, when popping the finalizer frame from the stack, the associated resource is closed again *before* the finalization statement s is run.

```
var file = open(path)
try { ... read(file) ... }
on resume { _ ⇒ open(path) }
finally { throw("abort"); close(file) }
```

In this program, assuming the handler for `exc` does not resume the continuation, `closeFile` is never actually run. Similarly, the updated rule *(suspend)* closes the resource during unwind mode when transitioning to reduction mode before executing the suspend clause. Dually, when rewinding, the rule *(resume')* re-opens the resource before running the resume clause. These invariants are not statically checked, but rather represent a set of best practices that users should follow.

When considering the instrumented abstract machine, this corresponds to the properties that $R(l_i) = 1$ for all labels l_i that are active on the stack and $R(l_j) = 0$ for all other labels in R. We define the predicate active in the following way:

$$\begin{aligned}
&\mathsf{active}(\bullet) && = \emptyset \\
&\mathsf{active}(\#_l\ \mathcal{F} :: E) && = \{\ l\ \} \cup \mathsf{active}(E) \\
&\mathsf{active}(\#_l\ \{\ \square\ \}\ \textbf{with}\ \{\ x \Rightarrow s\ \} :: E) && = \mathsf{active}(E) \\
&\mathsf{active}(F :: E) && = \mathsf{active}(E)
\end{aligned}$$

Notice that we do not consider labels contained in a # **unwind** or # **rewind** frame to be active. Instead, only finalizer frames add active resources.

For stating the resource safety theorem, we also define predicates ActiveOpen and InactiveClosed as shorthands for the propositions stating that all active resources should be open and all other resources should be closed.

$$\begin{aligned}
\mathsf{ActiveOpen}(M) &:= \forall l_i \in \mathsf{active}(M.E),\ M.R(l_i) = 1 \\
\mathsf{InactiveClosed}(M) &:= \forall l_i \in R \setminus \mathsf{active}(M.E),\ M.R(l_i) = 0
\end{aligned}$$

The extractor functions $M.R$ and $M.E$ on machines M, extracting the resource environment R and the stack E, respectively, are defined in the obvious way. With these definitions, we can formally state the theorem of resource safety as follows:

Theorem 3.6 (Resource Safety). *If* $\mathsf{ActiveOpen}(M)$ *and* $\mathsf{InactiveClosed}(M)$ *and* $M \longrightarrow M'$, *then* $\mathsf{ActiveOpen}(M')$ *and* $\mathsf{InactiveClosed}(M')$.

Proof. The proof proceeds by case analysis on the step $M \longrightarrow M'$. The only interesting cases are those in which finalizer frames are involved and either pushed onto the stack or from the stack onto the continuation. Notably, these cases fulfill the property of Resource Safety by construction. All the other cases are trivially true by assumption as no changes occur in R when taking a step. □

4 Implementation

As a proof-of-concept, we implemented the extensions of System $\Xi_{\not\to}$ relative to System Ξ for the Effekt language[1]. In this section, we briefly describe the Effekt language and discuss some of the interesting aspects of the implementation.

Effekt is a programming language [Brachthäuser et al. 2022; Brachthäuser et al. 2020] implementing features from both imperative and functional languages, such as algebraic data types, pattern matching, mutable state, regions and first-class functions. Most prominently, Effekt also supports lexical effect handlers as well as lightweight contextual effect polymorphism. The language comes with an online language tour, an online playground, and a Visual Studio Code extension.

Notably, Effekt compiles to several different backends, including LLVM, JavaScript and ChezScheme. We implemented the ideas presented in this paper solely for the JavaScript backend. The implementation closely follows the semantics given by the abstract machine semantics: we introduce a new kind of finalizer stack frame. Also, the pre-existing unwinding and rewinding process in the JavaScript runtime is augmented to account for the new finalizer stack frames. The changes to the Effekt compiler are straightforward and consist of about 500 changed lines of code.

[1] https://effekt-lang.org

4.1 Extended Example: Combining Backtracking and Resource Management

It is a very common use case for `try-finally` constructs to open a file handle and ensure its release through the `finally` clause. Furthermore, as we have seen, we can also use the ideas introduced by our work for achieving correct backtracking behavior of heap resources. In the following, we will showcase an example that combines both ideas and integrates nicely with the parser combinators presented in Section 2.

First, recall the `read` effect that reads a single character. Previously, we used this effect in conjunction with reading from an array and backtracking its position. Alternatively, we can actually directly read from a file.

```
def readFile(path: String) { program: (Ref[Int]) ⇒ Unit / read }: Unit = {
  val mode = ReadOnly()
  var file = open(path, mode)
  val position = ref(0)
  try {
    program(position)
  } with read {
    val p = position.get
    position.set(p + 1)
    resume(readChar(file, p))
  } on suspend { close(file); position.get }
  on resume { p ⇒ position.set(p); file = open(path, mode) }
  on return { _ ⇒ close(file) }
}
```

The function `readFile` receives a file path as an argument and a block `program` which in turn receives a global reference and yields `Unit`. Furthermore, `program` uses the `read` effect and requires the calling context to handle it. We start by opening a given file and storing the file handle in a mutable variable `file`. Additionally, we store the current position from which the next character is read in a global mutable reference `position`. In the try-body, we pass the `position` reference to `program` such that `program` has access to the current position, for example, for generating meaningful error messages during parsing. The handler of `read` advances the position by one and reads the next character from the file. Importantly, we use the `suspend`, `resume`, and `return` clauses for closing and opening the file for each stack unwind and rewind. We also backtrack the state of `position` by returning the current value from the `suspend` clause to the `resume` clause.

Most notably, we can use the `readFile` handler as a drop-in replacement for `feedGlobal` in our `parser` from Section 2, while still correctly managing the file handler in terms of resource safety.

```
def parse[R](file: String) { parser: () ⇒ R / Parser }: Option[R] / {} = {
  backtrack { option { readFile(file) { parser() } } }
}
```

The file is closed and re-opened on each backtracking attempt. A variation that only backtracks the position or moves the file cursor is possible. More problematic is the fact that the file is closed and opened for each effect operation that is used in `parser` but handled outside of `parse`. We discuss this issue in the next subsection.

4.2 Tail-Resumption Optimization

Effect handlers which are syntactically tail-resumptive — that is, handler implementations call `resume` exactly once and only in tail position — are optimized by the Effekt compiler such that the explicit unwinding and rewinding of the stack is not needed. Thus, intuitively, this optimization is applied to all programs that are of the following form (where ... does not mention k):

try { $f \Rightarrow s$ } **with** { $(x,\ k) \Rightarrow ...;\ k(v)$ } **def** $f(x)\ =\ \{ ...;$ **return** $v \};\ s$

The capability introduced by the handler's implementation is given the name `f`. The key insight is that resuming the continuation exactly once in tail position corresponds to a normal function return. Hence, the given program on the left is transformed to the optimized program on the right, where the `try` is removed and the handler is replaced by a simple function definition.

By applying this transformation, the effect call corresponds to a mere function call, rather than prompting the unwinding of the stack, yielding improved performance. However, since the finalization clauses rely on the explicit stack unwinding for triggering them, further care needs to be taken when employing this optimization with the requirement of preserving the same semantics.

```
try { f ⇒                          try {
  try { ...; f(v1); ... }            def f(x) = { ...; return v2 }
  on suspend { s2 }                  try { ...; f(v1); ... }
  on resume { x ⇒ s3 }               on suspend { s2 }
  on return { x ⇒ s4 }               on resume { x ⇒ s3 }
} with { (x, k) ⇒ ...; k(v2)         on return { x ⇒ s4 }
} on return { x ⇒ s5 }             } on return { x ⇒ s5 }
```

In the program on the left (before optimization), there is an outer tail-resumptive handler and an inner handler with finalization clauses. After optimizing the program, the original program is transformed to the program on the right such that the outer handler's implementation is extracted into the function definition `f` and only the `return` clause remains. Consequently, since each call to `f` no longer causes a stack unwinding in search of the needed handler, the `suspend` and `resume` clause are *not* executed when calling `f`. Thus, in general, this optimization is not semantics-preserving. As a solution, this optimization could either be disabled or the user can be given explicit control on when to apply it.

5 Performance Evaluation

To evaluate whether supporting the introduced finalization clauses adds significant overhead to the execution time of Effekt programs, we benchmark our implementation against an appropriate baseline. Furthermore, for investigating the performance penalty of disabling the tail-resumption optimization, we also benchmark our implementation against the baseline.

The benchmarks are adapted from a community-maintained benchmark suite [Hillerström et al. 2023]. It is important to note that none of these benchmarks use the finalization clauses, and we instead intend to measure the overhead of supporting finalization in Effekt's JavaScript runtime.

5.1 Benchmarking Methodology

The benchmarks were conducted using `hyperfine` version 1.19.0 [Peter 2023] on a machine with an Apple M1 Pro CPU with 8 cores and 16 GB of RAM, running macOS version 15.5. As a baseline, we used the latest commit in the Effekt compiler from which our implementation diverges (Effekt 0.14.0 at a4418db).

Table 1. Runtimes of the benchmarks in milliseconds. Lower is better. We compare two different groups: Our implementation with the tail-resumption optimization against the baseline (also with the same optimization enabled) and both implementations without the tail-resumption optimization, color-coded in *gray*.

	Effekt JS (finalizers)		Effekt JS (baseline)	
	w/ tail opt.	w/o tail opt.	w/ tail opt.	w/o tail opt.
product-early	110.56 ± 5.95	110.60 ± 6.61	111.26 ± 6.73	109.19 ± 0.84
nqueens	154.19 ± 7.99	152.13 ± 1.01	143.21 ± 6.57	141.34 ± 1.02
countdown	52.02 ± 4.49	241.78 ± 12.59	50.76 ± 0.59	221.85 ± 1.43
iterator	87.15 ± 0.61	313.40 ± 15.75	87.55 ± 6.32	288.59 ± 9.42
tree-explore	62.32 ± 0.86	62.84 ± 3.96	61.44 ± 1.11	62.09 ± 4.54
fibonacci-recursive	133.86 ± 1.24	135.44 ± 6.80	134.19 ± 1.31	134.87 ± 6.11
resume-nontail	1080.50 ± 17.87	1068.75 ± 18.93	1049.61 ± 10.24	1049.66 ± 19.06
triples	220.75 ± 1.47	223.82 ± 11.20	196.51 ± 1.33	198.94 ± 8.63
parsing-dollars	140.63 ± 1.01	362.57 ± 11.39	139.81 ± 0.80	313.71 ± 11.01
geo. mean slowdown	1.03	1.06	1.00	1.00

While the Effekt compiler has multiple backends (LLVM, ChezScheme, JavaScript), our implementation only targets the JavaScript backend using Node.js version 24.2.0.

For measuring only the runtime, we first generated the JavaScript files for each benchmark. We measured the time it takes to execute each generated JavaScript file using Node.js. Each benchmark is executed at least ten times, of which we obtain the arithmetic mean execution time and the standard deviation.

5.2 Benchmark Results

Table 1 shows the mean runtimes and standard deviations of the JavaScript backend using the effect-handlers benchmark suite [Hillerström et al. 2023].

Expectedly, turning off the tail-resumption optimization can result in considerable slowdowns. For example, the `countdown` benchmark is more than four times slower without said optimization. In the geometric mean, our implementation without optimization is 52% slower compared to our implementation with the tail-resumption optimization turned on.

Similarly, the baseline without optimization is 47% slower compared to the baseline with said optimization turned on, thus being slightly less negatively affected by turning off the optimization. Also, the optimization has no effect on benchmarks that do not feature tail-resumptive handlers, like `resume-nontail`, `fibonacci-recursive`, `tree-explore`, or `nqueens`, which is also expected.

Both without and with optimization, the baseline is almost always faster. Only for the benchmarks `product-early`, `fibonacci-recursive`, and `iterator`, our implementation is faster if the optimization are turned on. It should be noted that the majority of measurements overlap with respect to the range given by their standard deviation.

There are also cases where our implementation is notably slower than the baseline, like `triples`, where our implementation (with and without optimizations) is more than 12% slower compared to its respective baseline. Across all benchmarks, computing the geometric mean of the normalized mean execution times yields a mean slowdown of 3% with optimizations and 6% without optimizations compared to the respective baseline.

5.3 Discussion

Using finalization clauses introduces additional stack frames, as described in subsection 2.7. We suspect this to be a significant overhead as twice as many stack frames need to be unwound and rewound. To mitigate this overhead, we introduce the simple optimization of not pushing these frames if a handler has no `suspend`, `resume`, and `return` clause. Thus, since these benchmarks do not use finalization clauses, an equal amount of stack frames are pushed by both implementations. As our implementation is still slower in most of the benchmarks, we conjecture that the observed minor slowdowns can mainly be attributed to the added branching during unwinding and rewinding for checking for `suspend` and `resume` clauses.

6 Related Work

We review implementations of dynamic wind in the presence of both undelimited and delimited continuations. Then we discuss both dynamic and static approaches to resource management in the presence of effect handlers.

6.1 Resource Finalization and Exception Handlers

The standard `try-finally` construct is known from languages like Java, JavaScript, C#, and Python. However, since these languages do not support effect handlers — or, more generally, (delimited) continuations — it is not possible for exceptional computations to be resumed. Furthermore, in these languages, generators do not trigger finalization. Thus, finalization can be simplified and a construct like `on resume` is not necessary.

Resource-Acquisition-Is-Initialization (RAII) was originally introduced by C++ [Stroustrup 2013], and later adopted by other languages such as Rust [Klabnik and Nichols 2023], as a resource allocation strategy. It statically ties the lifetime of resources to that of the object allocating and owning them. In comparison, the ideas presented in this paper are largely orthogonal, as we mainly discuss control flow and present effect handlers augmented with constructs for dynamic resource finalization and backtracking.

Schuster et al. [2022] present a language with resource pools and lexical exception handlers together with an abstract machine semantics as well as a continuation-passing translation for it. They prove a resource safety theorem for the abstract machine semantics, and a simulation theorem for the continuation-passing translation. Whereas they use type-level regions, we use second-class blocks to ensure both resource safety and effect safety. We generalize their work from fixed to arbitrary cleanup actions to `suspend` clauses, that may themselves use resources and control effects, and more importantly from exception handlers to effect handlers, which necessitates rewinding and `resume` clauses.

6.2 Effect Parametricity and Finalization

de Vilhena and Pottier [2023a] discuss the impact of a language construct such as finalization on parametricity. Parametricity [Wadler 1989] (also known as the *abstraction theorem*, Reynolds [1983]) can be understood as interpreting a syntactic universal type as meta-level universal quantification over a given universe of semantic types [de Vilhena and Pottier 2023a]. Specifically, this means that parametric polymorphic functions must behave the same regardless of what type they are instantiated with.

Effect parametricity [Biernacki et al. 2017; Zhang and Myers 2019] extends the notion to parametric quantification over effects and is often understood as "handlers cannot interfere with parametric effects". de Vilhena and Pottier [2023a] describe how finalization interferes with effect parametricity.

While finalizers cannot change *how* an effect is handled, they can observe *that* an effect is handled. Consider the following program by de Vilhena and Pottier [2023b], translated to Effekt:

```
def observe { prog: => Unit / {} }: Int / {} = {
  var r = 0
  try { prog(); r } on suspend { r = r + 1 }
}
```

Due to *contextual effect polymorphism* [Brachthäuser et al. 2020], it is not possible that `observe` modifies the effect handlers in `prog`. However, by means of the `suspend` clause it can be observed whether (and how often) effects are used by `prog`. Pushing this even further, we can define a `catchAll` handler that prevents handling of any yet unhandled effect:

```
effect Raise(msg: String): Nothing
def catchAll[R] { prog: => R / {} }: R / { Raise } =
  try { prog() } on suspend { do Raise("not handling effect") }
```

Again, even though we do not mention any effects in the type of `prog` we want to handle, we can still handle all effects by just raising an exception that, for example, may abort the program. While interesting, we leave it to future work to study parametricity and finalization in a language like Effekt featuring contextual effect polymorphism instead of parametric quantification over effects.

6.3 Effect Handlers and Dynamic Finalization

Leijen [2018] proposes an extension of the Koka language [Leijen 2014] to address the interaction of effect handlers with external resources. In this extension, handlers can additionally contain `finally` and `initially` clauses. The `finally`-clause is executed when returning normally or when unwinding for non-resuming operations. Leijen recognizes that it is difficult to statically determine whether an effect operation actually resumes. Instead, they propose to manually `finalize` continuations, which resume with a special finalization exception that unwinds the continuation and invokes the finalizers contained therein. They conjecture but do not prove that all finalizers are executed. Koka also features a `return` operation that is triggered only if the handled program returns normally, mapping the returned result. Importantly, this is different from our `return` clause, which is executed both if the handled program returns normally or through the handler.

Sivaramakrishnan et al. [2021] describe the design and implementation of effect handlers for OCaml 5. To support the correct interaction with linear resources and to guarantee clean up, they require the continuation to be used linearly, but do not statically enforce this. A continuation needs to be either `continued`, or explicitly `discontinued`. The latter raises an exception at the call-site enabling exception handlers to free resources. OCaml 5 supports intercepting all effects and exceptions. In principle, on-suspend and on-resume clauses could be expressed this way. However, they state that "In the implementation, reperform is implemented as a primitive to avoid executing code on the resumption path".

The Eff language [Bauer and Pretnar 2015] also features `finally` clauses in effect handlers. It is semantically equivalent to the `on return` clause presented in this paper. Like `on return`, it can be thought of as an outer wrapper that applies a transformation to the result of the effect handler; either returning from the handled program or the handler [Bauer and Pretnar 2015]. To the best of our knowledge, there is no concept of `on suspend` and `on resume` in the Eff language.

Phipps-Costin et al. [2023] propose to extend WebAssembly with effect handlers. Here too, continuations are assumed to be used linearly and either need to be resumed or aborted explicitly. They introduce a specialized construct `resume_throw x h*` to manually resume a continuation exceptionally, triggering exception and handlers at the call-site.

6.4 Dynamic Wind and Control Operators

Friedman and Haynes [1985] introduce `dynamic-wind` with three nullary function arguments: `prelude`, `body`, and `postlude`. When no control operators are used, all three execute in order. However, whenever control-flow leaves the `body` through a use of the control operator `call/cc`, the `postlude` is executed. Conversely, whenever it re-enters, the `prelude` is executed again. While similar in spirit, our `suspend` clause is only executed when the body is re-entered, and `on resume` and `on return` allow us to distinguish between normal return and suspension.

Flatt et al. [2007] present an implementation of delimited control and, among other features, dynamic wind. Their semantics and implementation are constrained by the legacy behavior of Scheme, which did not directly support delimited control. In contrast, we can start from a clean slate. Moreover, our work is in the context of effect handlers taking into account delimited control. Their formalization is in an untyped setting, while we present a type- and effect-safe language. Moreover, we state and prove our theorem of resource safety.

Sitaram [2003] discuss how a more general variant of `unwind-protect` known from Lisp should behave in the presence of higher-order control, for example continuations. An unwind-protected program consists of the program itself and a postlude that is guaranteed to be executed when control flow exits the protected program. In programs without continuations, an already exited context cannot be re-entered, thus rendering the idea of prelude that is run upon re-entering obsolete. Sitaram argue that this should still be the case in the presence of continuations by ensuring the postlude is only invoked upon a final control flow exit and not for each continuation call, meaning that the postlude is executed exactly once.

Pitman [2003] agree with this sentiment and state that `dynamic-wind` is not the solution for this problem as the postlude and prelude is run on every exit and resumption, for example when working with a file.

Importantly, Clinger [2003] show that it is indeed possible to express `unwind-protect` in terms of `dynamic-wind`, ensuring resources are only cleaned-up once by disallowing the re-entry, thus only permitting one-shot continuations. However, Clinger point out that distinguishing between these two different control flow exits is non-trivial and the added complexity might not be worth it. Indeed, a continuation call may be arbitrarily complex in the amount of time its execution takes. It is conceivable that in such cases the temporary finalization of resources until the later resumption of the continuation may actually be beneficial.

6.5 Effect Handlers and Static Control-Flow Linearity

Brachthäuser and Leijen [2023] propose a type system that statically enforces control-flow linearity, thus ruling out undesirable interactions of control and resources. The way a handler uses a continuation is determined by a simple syntactic check.

Tang et al. [2024] track control-flow linearity and additionally integrate it into a full linear type system, yielding more precision.

Similarly, van Rooij and Krebbers [2025] present an affine type system and effect system for distinguishing between effects whose handlers use the continuation in one-shot (linear) and multi-shot manner.

Using these approaches, a higher-order function that offers access to a linear resource like a file could require the argument function to have linear control flow. Static approaches like these are orthogonal to our work: we offer runtime constructs that allow for transparently backtracking external state when possible, whereas their goal is to rule out bad interactions where it is not.

7 Conclusion

In this paper, we discussed and offered a solution for obtaining well-defined behavior for the combination of dynamic wind with lexical effect handlers. Specifically, resuming and capturing continuations in the suspend or resume clauses is well-behaved and can be used to achieve well-formed resource bracketing and backtracking of external state. This stands in contrast to earlier work where this is considered undefined behavior. Furthermore, we endowed System $\Xi_{\leftrightarrow}$ with a type system and fine-grained operational semantics and proved the soundness of the formal language System $\Xi_{\leftrightarrow}$ by showing the theorems of progress and preservation. Additionally, we offered an implicit usage guideline by instrumenting the abstract machine. When adhering to it, only active resources are available and all others have been released. We formalized this statement as resource safety and presented a proof. For evaluating our ideas, we implemented dynamic wind as described in this paper as an extension to the existing Effekt language. Notably, we discussed current limitations regarding the existing tail-resumption optimization applied in the Effekt compiler, as it is not necessarily semantics-preserving in the presence of finalization clauses. For addressing this concern, in the future, it would be interesting to complement our dynamic approach to resource management with static checking of control-flow linearity. We hope to achieve both: backtracking where it is possible and definitive finalization where it is not.

Data-Availability Statement

For evaluating our ideas, we implemented the ideas presented in this paper as an extension to the existing Effekt[2] compiler written in Scala. We submit this implementation as an artifact [Voigt et al. 2025], including a selected set of examples that can be run using the compiler as well as the used benchmark programs and results.

Acknowledgments

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

[2] https://github.com/effekt-lang/effekt

References

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. doi:10.1016/j.jlamp.2014.02.001

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages. doi:10.1145/3158096

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. *Proc. ACM Program. Lang.* 6, OOPSLA, Article 76 (apr 2022), 30 pages. doi:10.1145/3527320

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). doi:10.1145/3428194

Jonathan Immanuel Brachthäuser and Daan Leijen. 2023. *Qualified Effect Types – Taming Control-Flow through Linear Effect Handlers.* Technical Report MSR-TR-2023-42. Microsoft Research.

William D. Clinger. 2003. Implementation of `unwind-protect` in Portable Scheme. (2003). http://www.ccs.neu.edu/home/will/UWESC/uwesc.sch

Paulo Emílio de Vilhena and François Pottier. 2023a. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 225–252. doi:10.1007/978-3-031-30044-8_9

Paulo Emílio de Vilhena and François Pottier. 2023b. A Type System for Effect Handlers and Dynamic Labels. (2023). https://devilhena-paulo.github.io/files/tes-slides.pdf

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *Proceedings of the International Conference on Functional Programming* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/1291151.1291178

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining control. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 245–254. doi:10.1145/318593.318654

Daniel Hillerström, Filip Koprivec, and Philipp Schuster. 2023. *Effect handlers benchmarks suite.* https://github.com/effect-handlers/effect-handlers-bench

Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8, 4 (July 1998), 437–444.

Steve Klabnik and Carol Nichols. 2023. *The Rust Programming Language.* No Starch Press.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Proceedings of the Workshop on Mathematically Structured Functional Programming. *Electronic Proceedings in Theoretical Computer Science.* doi:10.4204/eptcs.153.8

Daan Leijen. 2016. *Algebraic Effects for Functional Programming.* Technical Report. MSR-TR-2016-29. Microsoft Research technical report.

Daan Leijen. 2017. Implementing Algebraic Effects in C. In *Proceedings of the Asian Symposium on Programming Languages and Systems.* Springer International Publishing, Cham, Switzerland, 339–363. doi:10.1007/978-3-319-71237-6_17

Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. Microsoft Research. 35 pages.

Paul Blain Levy. 2001. *Call-by-push-value.* Ph. D. Dissertation. Queen Mary and Westfield College, University of London. https://pblevy.github.io/papers/thesisqmwphd.pdf Research Report No. RR-01-03.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. doi:10.1145/73560.73564

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications.* ACM, New York, NY, USA, 234–251. doi:10.1145/3022671.2984009

David Peter. 2023. *hyperfine.* https://github.com/sharkdp/hyperfine

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. 7, OOPSLA2, Article 238 (oct 2023), 26 pages. doi:10.1145/3622814

Kent Pitman. 2003. Unwind-Protect versus Continuations. (2003). http://www.nhplace.com/kent/PFAQ/unwind-protect-vs-continuations-original.html

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming.* Springer-Verlag, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. doi:10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)..

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP World Computer Congress*. Elsevier (North-Holland), Amsterdam, The Netherlands, 513–523.

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2022. Region-based Resource Management and Lexical Exception Handlers in Continuation-Passing Style. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings* (Munich, Germany). Springer-Verlag, Berlin, Heidelberg, 492–519. doi:10.1007/978-3-030-99336-8_18

Dorai Sitaram. 2003. Unwind-protect in portable Scheme. In *Proceedings of the 4th Workshop on Scheme and Functional Programming* (2003-11-07) *(Tech. Rep., UUCS-03-023)*, Matthew Flatt (Ed.). 48–52.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

Bjarne Stroustrup. 2013. *The C++ programming language*. Pearson Education.

Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL, Article 54 (Jan. 2024), 29 pages. doi:10.1145/3632896

Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *Proc. ACM Program. Lang.* 9, POPL, Article 5 (Jan. 2025), 29 pages. doi:10.1145/3704841

David Voigt, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. *Dynamic Wind for Effect Handlers (Artifact)*. doi:10.5281/zenodo.16901700

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, NY, USA, 347–359.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. doi:10.1145/3290318

A Appendix

A.1 Proof: Resource Safety

For readability, we re-state the theorem and definitions.

$$
\begin{array}{lll}
\mathsf{active}(\bullet) & & = \emptyset \\
\mathsf{active}(\#_l\ \mathcal{F}\ ::\ E) & & = \{\ l\ \} \cup \mathsf{active}(E) \\
\mathsf{active}(\#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ x \Rightarrow s\ \}\ ::\ E) & & = \mathsf{active}(E) \\
\mathsf{active}(F\ ::\ E) & & = \mathsf{active}(E) \\
\\
\mathsf{active\_open}(M) & := \forall l_i \in \mathsf{active}(M.E),\ M.R(l_i) = 1 & \\
\mathsf{inactive\_closed}(M) & := \forall l_i \in R \setminus \mathsf{active}(M.E),\ M.R(l_i) = 0 &
\end{array}
$$

The extractor functions $M.R$ and $M.E$ are defined in the obvious way.

Theorem A.1 (Resource Safety). *If* $\mathsf{active\_open}(M)$ *and* $\mathsf{inactive\_closed}(M)$ *and* $M \longrightarrow M'$, *then* $\mathsf{active\_open}(M')$ *and* $\mathsf{inactive\_closed}(M')$.

Proof. By case analysis on $M \longrightarrow M'$.

Case return':

We have $M = \langle\ \mathbf{return}\ v \mid \#_l\ \mathcal{F}\ ::\ E \mid R\ \rangle$ and $M' = \langle\ s[x \mapsto v] \mid E' \mid R'\ \rangle$. We further know that $\mathsf{active}(E) \setminus \mathsf{active}(E') = \{\ l\ \}$ since the finalizer frame $\mathcal{F}$ is popped from the stack and thus now longer active. Therefore proving $\mathsf{inactive\_closed}(M')$ only amounts to showing that $R'(l) = 0$ which is trivially true by construction. By assumption we know that all other resources are closed and $\mathsf{active\_open}(M')$ is trivially true by assumption as well since $\mathsf{active}(E) \subseteq \mathsf{active}(E')$.

Case try:

We have $M = \langle\ \mathbf{try}\ \{\ f \Rightarrow s\ \}\ \mathbf{with}\ h \mid E\ \rangle$ and

$$M' = \langle\ s[f \mapsto \#_l\ \mathbf{cap}] \mid \#_l\ \{\ \square\}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ \mathcal{F}(h)\ ::\ E\ \rangle$$

We further know that $\mathsf{active}(E') \setminus \mathsf{active}(E) = \{\ l\ \}$ since a new finalizer frame is pushed onto the stack. Thus, we only need to show that $R'(l) = 1$, which is true by construction. The remainding proof goals follow by assumption.

Case suspend:

We have $M = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \#_{l'}\ \mathcal{F}\ ::\ E \looparrowright K \mid R\ \rangle$ and

$$M' = \langle\ \mathcal{F}.\mathbf{suspend} \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \#_{l'}\ \mathcal{F},\ K)\ ::\ E \mid R(l') = 0\ \rangle$$

Since again $\mathsf{active}(E) \setminus \mathsf{active}(E') = \{\ l\ \}$, we only need to show that $R'(l) = 0$ which is again true by construction. Showing $\mathsf{active\_open}(M')$ follows by assumption.

Case suspend':

We have $M = \langle\ \mathbf{return}\ v \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)\ ::\ E\ \rangle$ and

$$M' = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright (\mathcal{F},\ v)\ ::\ K\ \rangle$$

Since $\mathsf{active}(\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)\ ::\ E) = \mathsf{active}(E)$, the proof goals directly follow from assumption.

Case resume':

We have $M = \langle\ \mathbf{return}\ v \mid E \hookleftarrow (\mathcal{F},\ v)\ ::\ K\ \rangle$ and

$$M' = \langle\ \mathcal{F}.\mathbf{resume}[x \mapsto v] \mid \#\ \mathbf{rewind}(v,\ \mathcal{F},\ K)\ ::\ E\ \rangle$$

Since $\mathsf{active}(E) = \mathsf{active}(\#\ \mathbf{rewind}(v,\ \mathcal{F},\ K)\ ::\ E)$, the proof goals directly follow from assumption.

Case resume':

We have $M = \langle\ \mathbf{return}\ v \mid \#\,\mathbf{rewind}(v',\ \mathcal{F},\ K)\ ::\ E\ \rangle$ and $M' = \langle\ \mathbf{return}\ v' \mid \mathcal{F}\ ::\ E \hookleftarrow K\ \rangle$. Since $\mathsf{active}(E')\ \setminus\ \mathsf{active}(E) = \{\ l\ \}$, we only need to show that $R'(l) = 1$ for proving $\mathsf{active\_open}(M')$, which is true by construction. Showing $\mathsf{inactive\_closed}(M')$ follows by assumption since

$$R'\ \setminus\ \mathsf{active}(M')\ \subseteq R\ \setminus\ \mathsf{active}(M)$$

The remaining cases have that $\mathsf{active}(M) = \mathsf{active}(M')$ and no new labels are introduced to R, thus the proof goals directly follow from assumption. □

A.2 Abstract Machine Typing

For proving the theorems of progress and preservation, we need to add a type system to the abstract machine and its frames.

Figure 9 defines the rules for typing the abstract machine and continuations. We start by first discussing the block typing rules and then work our way towards the more complex stack and continuation typing rules.

For typing a capability of the form $\#_l\ \mathbf{cap}$ with rule B-Cap, we look up the label l in the label environment Ξ to find its annotated type. Typing a $\#\ \mathbf{resume}$ block with B-Resume is more involved. Given that the handler h introduces a capability of type $\overrightarrow{\tau_i} \to \tau_0$ and we can derive that the continuation K has the type $\tau_0 \to \tau$, we may conclude that $\#\ \mathbf{resume}(l,\ h,\ K)$ has the type $\tau_0 \to \tau_{rt}$. Notice that the continuation K expects a value of type τ_0, whereas the capability with the label l promises to yield a value of type τ_0. Due to the **return** clause in h with type $\tau \Rightarrow \tau_{rt}$, the result of resuming continuation is altered to yield a value of τ_{rt}.

Next, we turn to the frame typing rules. For checking the type of a frame F against a frame/stack type $\tau_1 \rightsquigarrow \tau_2$, we need to provide a label environment Ξ containing the labels of the capabilities in scope. Being the most simple, the rule F-Seq checks that a sequencing frame $\mathbf{val}\ x = \square;\ s$ has the type $\tau_1 \rightsquigarrow \tau_2$. For this conclusion to hold, we need to derive that s has the type τ_2 with $x :\ \tau_1$ in scope. The rule F-Unwind allows use to conclude that an unwind frame of the form $\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)$ has the type $\tau_{sp} \rightsquigarrow \tau_{rt}$. For that, we need type check the values $\overrightarrow{v}$ and ensure that the label l is bound in Ξ. Further, we must type check the finalizer $\mathcal{F}$ such that its type composes with the type of the continuation K. It is instructive to recall the semantics of the **unwind** frame. The $\#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)$ frame is pushed onto the runtime stack by the stepping rule *(suspend)* when a capability is called, the machine is in unwind mode and encounters a finalizer frame. The **unwind** frame serves as a reminder to later continue unwinding the stack onto the continuation K after pushing the handler frame and the result $v :\ \tau_{sp}$ of the **suspend** clause onto the continuation. Dually, the rule F-Rewind type checks a rewind frame $\#\ \mathbf{rewind}(v,\ \mathcal{F},\ K)$ against the type $\mathsf{Unit} \rightsquigarrow \tau_{rt}$. Recall that the **rewind** frame is the dual of the **unwind** frame. It is pushed onto the stack during rewind mode such that when encountered again after evaluating the **resume** clause, the machine is put into unwind mode again such that **return** v' is returned to the context where K is rewound onto the current stack E with the finalizer $\mathcal{F}$ pushed on top.

The machine typing rules $\vdash_m \langle\ s \mid E \mid K\ \rangle :\ \tau$ receive a machine and a stack type $\tau_1 \rightsquigarrow \tau_2$. The rule M-Normal checks the stack type of a machine $m = \langle\ s \mid E\ \rangle$ in reduction mode. Given that s is a value of type τ_1 and E has the stack type $\tau_1 \rightsquigarrow \tau_3$, $m :\ \tau_3$ can be concluded. For a machine $m = \langle\ s \mid E \looparrowright K\ \rangle$ in unwind mode, the rule M-Unwind allows us to conclude that $m :\ \tau_3$ given that $s :\ \tau_1$, the stack E has the type $\tau_2 \rightsquigarrow \tau_3$ and the continuation K has the type $K :\ \tau_1 \rightsquigarrow \tau_2$. Note that the types of the continuation and stack compose such that the result is of type τ_3. The rule M-Rewind is similar.

The rules $\vdash_{ctx} E :\ \tau \rightsquigarrow \tau \mid \Xi$ for typing a stack E against a stack type $\tau \rightsquigarrow \tau$ and a label environment containing the label of each handler present on the stack. The rule E-Empty types the empty stack, corresponding to the identity. E-Frame types a stack $F\ ::\ \Sigma$ with an arbitrary

stack frame F on top. Notice that the type of the frame $\tau_2 \rightsquigarrow \tau_3$ and that of the stack $\tau_1 \rightsquigarrow \tau_2$ compose, yielding the resulting type $\tau_1 \rightsquigarrow \tau_3$. For typing checking a stack E with a handler frame $\#_l \{ \square \}$ **with** $\{ (\overrightarrow{x},\ k) \Rightarrow s \}$ on top against $\tau_1 \rightsquigarrow \tau_2$, the rule E-Handler tells us that $E : \tau_1 \rightsquigarrow \tau_2$ and that the handler is of type τ_1 as well, while having $\overrightarrow{x : \tau_i}$ and the continuation $k : \tau_0 \to \tau_1$ in scope. Additionally, the label l is appended to the label environment Ξ as output.

The continuation typing rules $\Xi \vdash_{cnt} K : \tau \rightsquigarrow \tau \mid \Xi$ are very similar to the stack typing rules, except that they receive a label environment Ξ as input as well, corresponding to all the labels that are in scope on the stack the continuation is eventually rewound onto. The rules for typing the empty continuation $\bullet$ and the continuation $F :: K$ are is similar to the corresponding rules for stacks. C-Handler for typing continuation K with a handler on top is more interesting. First, we need to show that the remaining continuation has the type $\tau_1 \rightsquigarrow \tau_2$, given that the label l from the handler frame is in scope for the remainder of the continuation. We also need to show that the handler is well-typed similar to E-Handler. As a result, we may conclude that the overall type of the continuation is $\tau_1 \rightsquigarrow \tau_2$ with the label l added to the output label environment. Notice that compared to the E-Hanlder, the type of the language-level continuation k changed and has the same return type as the continuation K.

Block Typing. $\boxed{\Gamma \mid \Delta \mid \Xi \vdash\ b\ :\ \sigma}$

$$\frac{l :\ \overrightarrow{\tau_i} \to \tau_0\ \in \Xi}{\Gamma \mid \Delta \mid \Xi \vdash\ \#_l\ \mathbf{cap} :\ \overrightarrow{\tau_i} \to \tau_0}\ [\text{B-Cap}]$$

$$\frac{\Xi,\ l :\ \overrightarrow{\tau_i} \to \tau_0 \vdash_{cnt}\ K :\ \tau_0 \rightsquigarrow \tau \mid \Xi' \qquad \Gamma,\ \overrightarrow{x :\ \tau_i} \mid \Delta,\ k :\ \tau_0 \to \tau \mid \Xi \vdash\ s\ :\ \tau}{\Gamma \mid \Delta \mid \Xi \vdash\ \#\ \mathbf{resume}(\#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K) :\ \tau_0 \to \tau}\ [\text{B-Resume}]$$

Frame Typing. $\boxed{\Xi \vdash_{frm}\ F\ :\ \tau \rightsquigarrow \tau}$

$$\frac{x :\ \tau_1 \mid \bullet \mid \Xi \vdash\ s\ :\ \tau_2}{\Xi \vdash_{frm}\ \mathbf{val}\ x\ =\ \square;\ s\ :\ \tau_1 \rightsquigarrow \tau_2}\ [\text{F-Seq}]$$

$$\frac{\overrightarrow{\bullet \vdash\ v :\ \tau_i} \qquad l :\ \overrightarrow{\tau_i} \to \tau_0\ \in \Xi}{\Xi \vdash_{\mathcal{F}}\ \mathcal{F} :\ \tau \rightsquigarrow \tau_{rt} \mid \tau_{sp} \qquad \Xi \vdash_{cnt}\ K :\ \tau_0 \rightsquigarrow \tau \mid \Xi'}\ [\text{F-Unwind}]$$
$$\Xi \vdash_{frm}\ \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K)\ :\ \tau_{sp} \rightsquigarrow \tau_{rt}$$

$$\frac{\bullet \vdash\ v :\ \tau_0 \qquad \Xi \vdash_{\mathcal{F}}\ \mathcal{F} :\ \tau \rightsquigarrow \tau_{rt} \mid \tau_{sp} \qquad \Xi \vdash_{cnt}\ K :\ \tau_0 \rightsquigarrow \tau \mid \Xi'}{\Xi \vdash_{frm}\ \#\ \mathbf{rewind}(v,\ \mathcal{F},\ K)\ :\ \mathsf{Unit} \rightsquigarrow \tau_{rt}}\ [\text{F-Rewind}]$$

Finalizer Typing. $\boxed{\Xi \vdash_{\mathcal{F}}\ \mathcal{F}\ :\ \tau \rightsquigarrow \tau \mid \tau}$

$$\frac{\bullet \mid \bullet \mid \Xi \vdash\ s_{sp}\ :\ \tau_{sp} \qquad x :\ \tau_{sp} \mid \bullet \mid \Xi \vdash\ s_{rs} :\ \mathsf{Unit} \qquad x :\ \tau \mid \bullet \mid \Xi \vdash\ s_{rt}\ :\ \tau_{rt}}{\Xi \vdash_{\mathcal{F}}\ \{\ \square\ \}\ \mathbf{on\ suspend}\ \{\ s_{sp}\ \}\ \mathbf{on\ resume}\ \{\ x \Rightarrow s_{rs}\ \}\ \mathbf{on\ return}\ \{\ x \Rightarrow s_{rt}\ \}\ :\ \tau \rightsquigarrow \tau_{rt} \mid \tau_{sp}}\ [\text{F-Finalizer}]$$

Machine Typing. $\boxed{\vdash_m\ M\ :\ \tau}$

$$\frac{\bullet \mid \bullet \mid \Xi \vdash\ s :\ \tau_1 \qquad \vdash_{ctx}\ E :\ \tau_1 \rightsquigarrow \tau_3 \mid \Xi}{\vdash_m\ \langle\ s \mid E\ \rangle\ :\ \tau_3}\ [\text{M-Reduction}]$$

$$\frac{\bullet \mid \bullet \mid \Xi \vdash\ \#_l\ \mathbf{cap}(\overrightarrow{v}) :\ \tau_1 \qquad \Xi \vdash_{cnt}\ K :\ \tau_1 \rightsquigarrow \tau_2 \mid \Xi' \qquad \vdash_{ctx}\ E :\ \tau_2 \rightsquigarrow \tau_3 \mid \Xi}{\vdash_m\ \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright K\ \rangle\ :\ \tau_3}\ [\text{M-Unwind}]$$

$$\frac{\bullet \mid \bullet \mid \Xi \vdash\ \mathbf{return}\ v :\ \tau_1 \qquad \Xi \vdash_{cnt}\ K :\ \tau_1 \rightsquigarrow \tau_2 \mid \Xi' \qquad \vdash_{ctx}\ E :\ \tau_2 \rightsquigarrow \tau_3 \mid \Xi}{\vdash_m\ \langle\ \mathbf{return}\ v \mid E \hookleftarrow K\ \rangle\ :\ \tau_3}\ [\text{M-Rewind}]$$

Fig. 9. Typing rules for the runtime blocks, frames and abstract machine of System $\Xi_{\looparrowright}$

Stack Typing. $\boxed{\vdash_{ctx} E \;:\; \tau \rightsquigarrow \tau \mid \Xi}$

$$\frac{}{\vdash_{ctx} \bullet \;:\; \tau \rightsquigarrow \tau \mid \emptyset}\;[\textsc{E-Empty}] \qquad \frac{\Xi \vdash_{frm} F : \tau_1 \rightsquigarrow \tau_2 \qquad \vdash_{ctx} E : \tau_2 \rightsquigarrow \tau_3 \mid \Xi}{\vdash_{ctx} F :: E \;:\; \tau_1 \rightsquigarrow \tau_3 \mid \Xi}\;[\textsc{E-Frame}]$$

$$\frac{\vdash_{ctx} E : \tau_1 \rightsquigarrow \tau_2 \mid \Xi \qquad \overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_1 \mid \Xi \vdash s : \tau_1}{\vdash_{ctx} \#_l \{ \square \}\ \textbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \} :: E \;:\; \tau_1 \rightsquigarrow \tau_2 \mid \Xi, l : \overrightarrow{\tau_i} \to \tau_0}\;[\textsc{E-Handler}]$$

$$\frac{\vdash_{ctx} E : \tau_{rt} \rightsquigarrow \tau_2 \mid \Xi \qquad \Xi \vdash_{\mathcal{F}} \mathcal{F} : \tau_1 \rightsquigarrow \tau_{rt} \mid \tau_{sp}}{\vdash_{ctx} \mathcal{F} :: E : \tau_1 \rightsquigarrow \tau_2 \mid \Xi}\;[\textsc{E-Finalizer}]$$

Continuation Typing. $\boxed{\Xi \vdash_{cnt} K \;:\; \tau \rightsquigarrow \tau \mid \Xi}$

$$\frac{}{\Xi \vdash_{cnt} \bullet \;:\; \tau \rightsquigarrow \tau \mid \emptyset}\;[\textsc{K-Empty}] \qquad \frac{\Xi \vdash_{frm} F : \tau_2 \rightsquigarrow \tau_3 \qquad \Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'}{\Xi \vdash_{cnt} F :: K \;:\; \tau_1 \rightsquigarrow \tau_3 \mid \Xi'}\;[\textsc{K-Frame}]$$

$$\frac{\Xi, l : \overrightarrow{\tau_i} \to \tau_0 \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi' \qquad \overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_2 \mid \Xi \vdash s : \tau_2}{\Xi \vdash_{cnt} \#_l \{ \square \}\ \textbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \} :: K \;:\; \tau_1 \rightsquigarrow \tau_2 \mid \Xi', l : \overrightarrow{\tau_i} \to \tau_0}\;[\textsc{K-Handler}]$$

$$\frac{\Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi' \qquad \Xi \vdash_{\mathcal{F}} \mathcal{F} : \tau_2 \rightsquigarrow \tau_{rt} \mid \tau_{sp} \qquad \bullet \vdash v : \tau_{sp}}{\Xi \vdash_{cnt} (\mathcal{F}, v) :: K : \tau_1 \rightsquigarrow \tau_{rt} \mid \Xi'}\;[\textsc{K-Finalizer}]$$

Fig. 10. Typing rules of the stack E and continuation K of the abstract machine of System $\Xi_{\not\to}$

A.3 Proof: Progress

For completness, we restate the Progress Theorem 3.3.

Theorem A.2 (Progress). *If* $\vdash_m m : \tau$, *then there either exists a value* v *such that* $m = \langle\ \textbf{return}\ v \mid \bullet\ \rangle$ *or a machine* m' *with* $m \longrightarrow m'$.

Proof. By case distinction on the derivation of $\vdash_m : \tau$:

Case M-Normal:

We are given $\bullet \mid \bullet \mid \Xi \vdash s : \tau_1$ and $\vdash_{ctx} E : \tau_1 \rightsquigarrow \tau_3 \mid \Xi$. By case distinction on the typing derivation of $\bullet \mid \bullet \mid \Xi \vdash s : \tau_1$:

Sub-Case Val:

Apply *(push)*.

Sub-Case Ret:

We have $s = \textbf{return}\ v$. By case distinction on E:

Sub-Sub-Case $E =:$

Immediate since $m = \langle\ \textbf{return}\ v \mid \bullet\ \rangle$.

Sub-Sub-Case $E = \mathcal{F} ::: E'$:

Apply *(return')*.

Sub-Sub-Case $E = \#_l\{\bullet\}$ with $\{(\overrightarrow{x}, k) \Rightarrow stmt\} ::: E'$:

Apply *(return)*.

Sub-Sub-Case $E = \#_l unwind(\overrightarrow{v}, \mathcal{F}, K) ::: E'$:

Apply *(suspend')*.

Sub-Sub-Case $E = \#rewind(v', \mathcal{F}, K) ::: E'$:

Apply *(resume')*.

Sub-Case App:
We have $s = b$. By case distinction on b:
Sub-Sub-Case $b = f$:
Contradiction since $\Delta = \bullet$.
Sub-Sub-Case $b = \{(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow stmt\}$:
Apply *(cong)* with *(app)*.
Sub-Sub-Case $b = \#cap(\overrightarrow{v})$:
Apply *(cap)*
Sub-Sub-Case $b = \#resume(l, \mathcal{F}, K)$:
Apply *(resume)*.
Sub-Case Def:
We have $\bullet \mid \bullet \mid \Xi \vdash b : \sigma$ and $\bullet \mid f : \sigma \mid \Xi \vdash s : \tau'$. If $b = f$ we have a contradiction since $\Delta = \bullet$. Otherwise, we may apply *(cong)* with *(def)*.
Case M-Unwind:
We know

(1) $m = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright K\ \rangle$
(2) $\bullet \mid \bullet \mid \Xi \vdash \#_l\ \mathbf{cap}(\overrightarrow{v}) : \tau_1$
(3) $\Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2$
(4) $\vdash_{ctx} E : \tau_2 \rightsquigarrow \tau_3 \mid \Xi$

By case distinction on E:
Sub-Case $E = \bullet$:
Contradiction, since by inversion on (4), we know that $\Xi = \bullet$ but then (2) cannot hold.
Sub-Case $E = \mathcal{F} ::: E'$:
Apply *(suspend)*
Sub-Case $E = \#_{l'}\{\bullet\}$ with $\{(\overrightarrow{x}, k) \Rightarrow stmt\} ::: E'$:
If $l = l'$ apply *(handle)*, otherwise *(unwind')*
Sub-Case $E = F :: E'$:
Apply *(unwind)*.
Case M-Rewind:
We have

(1) $m = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright K\ \rangle$
(2) $\bullet \mid \bullet \mid \Xi \vdash \#_l\ \mathbf{cap}(\overrightarrow{v}) : \tau_1$
(3) $\Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2$
(4) $\vdash_{ctx} E : \tau_2 \rightsquigarrow \tau_3 \mid \Xi$

By case distinction on K:
Sub-Case $K = \bullet$:
Appply *(stop)*
Sub-Case $K = \mathcal{F} ::: K'$:
Apply *(resume')*
Sub-Case $K = \#_{l'}\{\bullet\}$ with $\{(\overrightarrow{x}, k) \Rightarrow stmt\} ::: K'$:
(rewind')
Sub-Case $E = F :: E'$:
Apply *(rewind)*. □

A.4 Proof: Preservation

We need some standard auxiliary lemmas regarding the substitution of values and blocks.

Lemma A.3 (Substitution Lemma - Values).

(1) Given a statement $\Gamma,\ x:\ \tau_1 \mid \Delta \mid \Xi \vdash\ \ s:\ \tau_2$ *and a value* $\Gamma \vdash\ \ v:\ \tau_1$*, then* $\Gamma \mid \Delta \mid \Xi \vdash\ \ s[x \mapsto v]:\ \tau_2$.

(2) Given an expression $\Gamma,\ x:\ \tau_1 \vdash\ \ e:\ \tau_2$ *and a value* $\Gamma \vdash\ \ v:\ \tau_1$*, then* $\Gamma \mid \Delta \mid \Xi \vdash\ \ e[x \mapsto v]:\ \tau_2$.

(3) Given a block $\Gamma,\ x:\ \tau_1 \mid \Delta \mid \Xi \vdash\ \ b:\ \tau_2$ *and a value* $\Gamma \vdash\ \ v:\ \tau_1$*, then* $\Gamma \mid \Delta \mid \Xi \vdash\ \ b[x \mapsto v]:\ \tau_2$.

Lemma A.4 (Substitution Lemma - Blocks).

(1) Given a statement $\Gamma \mid \Delta,\ f:\ \sigma \mid \Xi \vdash\ \ s:\ \tau$ *and a block* $\Gamma \mid \Delta \mid \Xi \vdash\ \ b:\ \sigma$*, then* $\Gamma \mid \Delta \mid \Xi \vdash\ \ s[f \mapsto\ ]:\ \tau$.

(2) Given a block $\Gamma \mid \Delta,\ f:\ \sigma_2 \mid \Xi \vdash\ \ b_1:\ \sigma_1$ *and a block* $\Gamma \mid \Delta \mid \Xi \vdash\ \ b_2:\ \sigma_2$*, then* $\Gamma \mid \Delta \mid \Xi \vdash\ \ b_1[f \mapsto b_2]:\ \sigma_1$.

We restate the Preservation Theorem 3.4 for better readability.

Theorem A.5 (Preservation). *If* $\vdash_m\ \ m:\ \tau$ *and there exists* $m \longrightarrow m'$*, then* $\vdash_m\ \ m':\ \tau$.

Proof. By case analysis on the derivation of $\vdash_m\ \ m:\ \tau$, followed by a case analysis on $m \longrightarrow m'$.

Case M-Reduction:

By case anaylsis on $m \longrightarrow m'$:

Sub-Case pop:

We know

(1) $m\ = \langle\ \mathbf{return}\ v \mid \mathbf{val}\ x\ =\ \square;\ s\ ::\ E\ \rangle:\ \tau$

(2) $m'\ = \langle\ s[x \mapsto v] \mid E\ \rangle$

By inversion on (1), we get

(3) $\bullet \mid \bullet \mid \Xi \vdash\ \ \mathbf{return}\ v:\ \tau_1$

(4) $\vdash_{ctx}\ \mathbf{val}\ x\ =\ \square;\ s\ ::\ E:\ \tau_1 \rightsquigarrow \tau_2 \mid \Xi$

By inversion on (3), we know

(5) $\bullet \vdash\ \ v:\ \tau_1$

By inversion on (4)

(6) $\Xi \vdash_{frm}\ \mathbf{val}\ x\ =\ \square;\ s:\ \tau_1 \rightsquigarrow \tau_2$

(7) $\vdash_{ctx}\ E:\ \tau_2 \rightsquigarrow \tau$

By inversion on (6)

(8) $x:\ \tau_1 \mid \bullet \mid \Xi \vdash\ \ s:\ \tau_2$

By Lemma A.3 on (5) and (6)

(9) $\bullet \mid \bullet \mid \Xi \vdash\ \ s[x \mapsto v]:\ \tau_2$

Finally, by M-Reduction applied to (7), (9), we get the desired result.

Sub-Case push:

We know by assumption

(1) $m\ = \langle\ \mathbf{val}\ x\ =\ s_1;\ s_2 \mid E\ \rangle:\ \tau$

(2) $m'\ = \langle\ s_1 \mid \mathbf{val}\ x\ =\ \square;\ \sigma_2\ ::\ E\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash\ \ \mathbf{val}\ x\ =\ s_1;\ s_2\ :\ \tau_1$

(4) $\vdash_{ctx}\ E:\ \tau_1 \rightsquigarrow \tau \mid \Xi$

By inversion on (3)

(5) $\bullet \mid \bullet \mid \Xi \vdash \ s_1 \ : \ \tau_3$
(6) $x : \ \tau_3 \mid \bullet \mid \Xi \vdash \ s_2 \ : \ \tau_1$

Apply F-Seq on (3)

(7) $\Xi \vdash_{frm} \mathbf{val}\ x \ = \ \square; \ s \ : \ \tau_3 \rightsquigarrow \tau_1$

Apply E-Frame to (4) and (7)

(8) $\vdash_{ctx} \mathbf{val}\ x \ = \ \square; \ s \ :: \ E \ : \ \tau_3 \rightsquigarrow \tau$

Apply M-Reduction to (5) and (8) to get the desired result.

Sub-Case return:

By assumption, we know

(1) $m \ = \langle \ \mathbf{return}\ v \mid \#_l \{ \square \}\ \mathbf{with}\ \{ \ (\overrightarrow{x}, \ k) \Rightarrow s \ \} \ :: \ E \ \rangle : \ \tau$
(2) $m' \ = \langle \ \mathbf{return}\ v \mid E \ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \mathbf{return}\ v : \ \tau_1$
(4) $\vdash_{ctx} \ \#_l \{ \square \}\ \mathbf{with}\ \{ \ (\overrightarrow{x}, \ k) \Rightarrow s \ \} \ :: \ E : \ \tau$

By inversion on (4)

(5) $\vdash_{ctx} \ E : \ \tau_1 \rightsquigarrow \tau \mid \Xi$
(6) $\overrightarrow{x : \tau_i} \mid k : \ \tau_0 \rightarrow \tau \mid \Xi \vdash \ s : \ \tau_1$

By applying [M-Reduction] to (3) and (5) we get the desired result.

Sub-Case return':

By assumption, we know

(1) $m \ = \langle \ \mathbf{return}\ v \mid \mathcal{F} \ :: \ E \ \rangle : \ \tau$
(2) $m' \ = \langle \ s_{rt}[x \mapsto v] \mid E \ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \mathbf{return}\ v : \ \tau_1$
(4) $\vdash_{ctx} \ \mathcal{F} \ :: \ E : \ \tau_1 \rightsquigarrow \tau$

By inversion on (4)

(5) $\vdash_{ctx} \ E : \ \tau_{rt} \rightsquigarrow \tau \mid \Xi$
(6) $\Xi \vdash_{\mathcal{F}} \mathcal{F} : \ \tau_1 \rightsquigarrow \tau_{rt} \mid \tau_{sp}$

By inversion on (6)

(7) $\bullet \mid \bullet \mid \Xi \vdash \ s_{sp} : \ \tau_{sp}$
(8) $x : \ \tau_{sp} \mid \bullet \mid \Xi \vdash \ s_{rs} : \ \mathsf{Unit}$
(9) $x : \ \tau_1 \mid \bullet \mid \Xi \vdash \ s_{rt} : \ \tau_{rt}$

By inversion on (3)

(10) $\bullet \vdash \ v : \ \tau_1$

By Lemma A.3 with (9) and (10)

(11) $\bullet \mid \bullet \mid \Xi \vdash \ s_{rt}[x \mapsto v] : \ \tau_{rt}$

By [M-Reduction] applied to (5) and (11) we get the desired result.

Sub-Case try:

By assumption, we know

(1) $m \ = \langle \ \mathbf{try}\ \{ f \Rightarrow s \}\ \mathbf{with}\ h \mid E \ \rangle : \ \tau$
(2) $m' \ = \langle \ s[f \mapsto \#_l\ \mathbf{cap}] \mid \#_l \{ \square \}\ \mathbf{with}\ \{ \ (\overrightarrow{x}, \ k) \Rightarrow s \ \} \ :: \ \{ \square \}\, \mathcal{F}(h) \ :: \ E \ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \mathbf{try}\ \{ f \Rightarrow s \}\ \mathbf{with}\ h \ : \ \tau_1$

(4) $\vdash_{ctx} E : \tau_1 \rightsquigarrow \tau \mid \Xi$

By inversion on (3)

(5) $\bullet \mid f : \overrightarrow{\tau_i} \to \tau_0 \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0 \vdash s : \tau_2$

(6) $\bullet \mid \bullet \mid \Xi \vdash h : \overrightarrow{\tau_i} \to \tau_0 \mid \tau_2 \Rightarrow \tau_1 \mid \tau_{sp}$

By inversion on (6)

(7) $\overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_2 \mid \Xi \vdash s_h : \tau_2$

(8) $\bullet \mid \bullet \mid \Xi \vdash s_{sp} : \tau_{sp}$

(9) $x : \tau_{sp} \mid \bullet \mid \Xi \vdash s_{rs} : \mathsf{Unit}$

(10) $x : \tau_2 \mid \bullet \mid \Xi \vdash s_{rt} : \tau_1$

By using [B-Cap], we get

(11) $\bullet \mid \bullet \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0 \vdash \#_l\ \mathbf{cap} : \overrightarrow{\tau_i} \to \tau_0$

By applying Lemma A.4 to (5) with (11), we get

(12) $\bullet \mid \bullet \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0 \vdash s[f \mapsto \#_l\ \mathbf{cap}] : \tau_2$

By applying F-Finalizer to (8), (9) and (10)

(13) $\Xi \vdash_{\mathcal{F}} \mathcal{F} : \tau_2 \rightsquigarrow \tau_1 \mid \tau_{sp}$

By applying E-Finalizer to (4) and (13)

(14) $\vdash_{ctx} \mathcal{F} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi$

By applying E-Handler to (7) and (14)

(15) $\vdash_{ctx} \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s_h\ \} :: \mathcal{F} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0$

By applying [M-Reduction] to (12) and (15), we get the desired result.

Sub-Case CAP:

(1) $m = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E\ \rangle : \tau$

(2) $m' = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright \bullet\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \#_l\ \mathbf{cap}(\overrightarrow{v}) : \tau_1$

(4) $\vdash_{ctx} E : \tau_1 \rightsquigarrow \tau \mid \Xi$

By K-Empty

(5) $\Xi \vdash_{cnt} \bullet : \tau_1 \rightsquigarrow \tau_1$

Then by [M-Unwind] applied to (3), (4) and (5), we get the desired result.

Sub-Case SUSPEND':

We have

(1) $m = \langle\ \mathbf{return}\ v \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) :: E\ \rangle : \tau$

(2) $m' = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright (\mathcal{F},\ v) :: K\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \mathbf{return}\ v : \tau_1$

(4) $\vdash_{ctx} \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) :: E : \tau_1 \rightsquigarrow \tau \mid \Xi$

By inversion on (3)

(5) $\bullet \vdash v : \tau_1$

By inversion on (4)

(6) $\vdash_{ctx} E : \tau_{rt} \rightsquigarrow \tau \mid \Xi$

(7) $\Xi \vdash_{frm} \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) : \tau_{sp} \rightsquigarrow \tau_{rt}$

By inversion on (7)

(8) $\overrightarrow{\bullet \vdash v : \tau_i}$
(9) $l : \overrightarrow{\tau_i} \to \tau_0 \in \Xi$
(10) $\Xi \vdash_{\mathcal{F}} \mathcal{F} : \tau' \rightsquigarrow \tau_{rt} \mid \tau_{sp}$
(11) $\Xi \vdash_{cnt} K : \tau_0 \rightsquigarrow \tau' \mid \Xi'$

Applying K-Finlizer to (5), (10) and (11)

(12) $\Xi \vdash_{cnt} (\mathcal{F}, v) :: K : \tau_0 \rightsquigarrow \tau_{rt} \mid \Xi$
(13) $\bullet \mid \bullet \mid \Xi \vdash \#_l \, \mathbf{cap} : \overrightarrow{\tau_i} \to \tau_0$

Applying App to (8) and (13) yields

(14) $\bullet \mid \bullet \mid \Xi \vdash \#_l \, \mathbf{cap}(\overrightarrow{v}) : \tau_0$

Finally, applying M-Unwind to (6), (12) and (14) yields the desired result.

Sub-Case resume':

We know

(1) $m = \langle \, \mathbf{return}\ v \mid \#\, \mathbf{rewind}(v', \mathcal{F}, K) :: E \, \rangle : \tau$
(2) $m' = \langle \, \mathbf{return}\ v' \mid \mathcal{F} :: E \hookleftarrow K \, \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \mathbf{return}\ v : \tau_1$
(4) $\vdash_{ctx} \#\, \mathbf{rewind}(v', \mathcal{F}, K) :: E : \tau_1 \rightsquigarrow \tau \mid \Xi$

By inversion on (4)

(5) $\Xi \vdash_{frm} \#\, \mathbf{rewind}(v', \mathcal{F}, K) : \mathsf{Unit} \rightsquigarrow \tau_{rt}$
(6) $\vdash_{ctx} E : \tau_{rt} \rightsquigarrow \tau \mid \Xi$

By inversion on (5)

(7) $\bullet \vdash v' : \tau_0$
(8) $\Xi \vdash_{\mathcal{F}} \mathcal{F} : \tau' \rightsquigarrow \tau_{rt} \mid \tau_{sp}$
(9) $\Xi \vdash_{cnt} K : \tau_0 \rightsquigarrow \tau' \mid \Xi'$

Apply E-Finalizer to (6) and (8)

(10) $\vdash_{ctx} \mathcal{F} :: E : \tau' \rightsquigarrow \tau$

Apply Ret to

(11) $\bullet \mid \bullet \mid \Xi \vdash \mathbf{return}\ v : \tau_0$

Apply M-Rewind to (9), (10) and .

Sub-Case cont:

We have

(1) $m = \langle \, \mathbf{resume}(\#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \}, K)(v) \mid E \, \rangle : \tau$
(2) $m' = \langle \, \mathbf{return}\ v \mid \#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \} \hookleftarrow K \, \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \#\, \mathbf{resume}(\#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \}, K)(v) : \tau_1$
(4) $\vdash_{ctx} E : \tau_1 \rightsquigarrow \tau \mid \Xi$

By inversion on (3)

(5) $\bullet \mid \bullet \mid \Xi \vdash \#\, \mathbf{resume}(\#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x}, k) \Rightarrow s \}, K) : \tau_0 \to \tau_1$
(6) $\bullet \mid \bullet \mid \Xi \vdash v : \tau_0$

By inversion on (5)

(7) $\Xi, l : \overrightarrow{\tau_i} \to \tau_0 \vdash_{cnt} K : \tau_0 \rightsquigarrow \tau_1 \mid \Xi'$
(8) $\overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_1 \mid \Xi \vdash s : \tau_1$

By applying E-Handler to (4) and (8)

(9) $\vdash_{ctx} \#_l \{ \square \}$ **with** $\{ (\overrightarrow{x}, k) \Rightarrow s \} :: E : \tau_1 \rightsquigarrow \tau \mid \Xi, l : \overrightarrow{\tau_i} \to \tau_0$

By applying Ret to (6)

(10) $\bullet \mid \bullet \mid \Xi \vdash$ **return** $v : \tau_0$

By applying M-Rewind to (7), (9) and (10), we get the desired result.

Case M-Unwind:

Sub-Case unwind:

We know by assumption

(1) $m = \langle \#_l \mathbf{cap}(\overrightarrow{v}) \mid F :: E \looparrowright K \rangle : \tau$

(2) $m' = \langle \#_l \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright F :: K \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \#_l \mathbf{cap}(\overrightarrow{v}) : \tau_1$

(4) $\vdash_{ctx} F :: E : \tau_2 \rightsquigarrow \tau \mid \Xi$

(5) $\Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

By inversion (4)

(6) $\Xi \vdash_{frm} F : \tau_2 \rightsquigarrow \tau'$

(7) $\vdash_{ctx} E : \tau' \rightsquigarrow \tau \mid \Xi$

By K-Frame on (5) and (6)

(8) $\Xi \vdash_{cnt} F :: K : \tau_1 \rightsquigarrow \tau' \mid \Xi'$

Thus, by M-Unwind on (3), (7) and (8) we have

(10) $\langle \# \mathbf{cap}(\overrightarrow{v}) \mid E \looparrowright F :: K \rangle : \tau$

Sub-Case unwind':

We know by assumption

(1) $m = \langle \#_l \mathbf{cap}(\overrightarrow{v}) \mid \#_{l'} \{ \square \}$ **with** $\{ (\overrightarrow{x}, k) \Rightarrow s \} :: E \looparrowright K \rangle : \tau$

(2) $m' = \langle \#_l \mathbf{cap}(\overrightarrow{v} \mid E \looparrowright \#_{l'} \{ \square \}$ **with** $\{ (\overrightarrow{x}, k) \Rightarrow s \} :: K \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \#_l \mathbf{cap}(\overrightarrow{v}) : \tau_1$

(4) $\Xi \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

(5) $\vdash_{ctx} \#_{l'} \{ \square \}$ **with** $\{ (\overrightarrow{x}, k) \Rightarrow s \} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi$

By inversion on (5)

(6) $\vdash_{ctx} E : \tau_2 \rightsquigarrow \tau \mid \Xi \setminus l'$

(7) $\overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_2 \mid \Xi \setminus l' \vdash s : \tau_2$

By K-Handler on (4) and (7)

(8) $\Xi \setminus l' \vdash_{cnt} \#_{l'} \{ \square \}$ **with** $\{ (\overrightarrow{x}, k) \Rightarrow s \} :: K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi, l' : \overrightarrow{\tau_i} \to \tau_0$

By inversion on (3)

(9) $\bullet \mid \bullet \mid \Xi \vdash \#_l \mathbf{cap} : \overrightarrow{\tau_j} \to \tau_1$

(10) $\overrightarrow{\bullet \vdash v : \tau_j}$

By inversion on (9)

(11) $l \in \Xi$

(12) $l \in \Xi \setminus l'$

Thus by B-Cap on (12)

(13) $\bullet \mid \bullet \mid \Xi \setminus l' \vdash \#_l \mathbf{cap} : \overrightarrow{\tau_j} \to \tau_1$

By App applied to (10) and (13)

(14) $\bullet \mid \bullet \mid \Xi \setminus l \vdash \ \#_l\ \mathbf{cap}(\overrightarrow{v_j}) : \tau_1$

Finally, by M-Unwind applied to (6), (8) and (6) we get the desired result.

Sub-Case HANDLE:

We know

(1) $m = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \} :: E \hookrightarrow K\ \rangle : \tau$

(2) $m' = \langle\ s[\overrightarrow{x \mapsto v},\ k \mapsto \mathbf{resume}(\ \#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)] \mid E\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \#_l\ \mathbf{cap}(\overrightarrow{v}) : \tau_1$

(4) $\vdash_{ctx}\ \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi$

(5) $\Xi \vdash_{cnt}\ K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

By inversion on (4)

(6) $\vdash_{ctx}\ E : \tau_2 \rightsquigarrow \tau \mid \Xi \setminus l$

(7) $\overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_2 \mid \Xi \setminus l \vdash \ s : \tau_2$

By applying B-Resume to (5) and (7)

(8) $\bullet \mid \bullet \mid \Xi \setminus l \vdash \ \#\ \mathbf{resume}(\#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K) : \tau_0 \to \tau_2$

By inversion on (3)

(9) $\overrightarrow{\bullet \vdash \ v : \tau_i}$

By Lemma A.4 on (7), Resume and Lemma [#lm-subst-value] on (7), (9)

(10) $\bullet \mid \bullet \mid \Xi \setminus l \vdash \ s[\overrightarrow{x \mapsto v},\ k \mapsto \mathbf{resume}(\ \#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \},\ K)] : \tau_2$

Finally, by applying E-Reduction to (6) and (10), we get the desired result.

Sub-Case SUSPEND:

We know

(1) $m = \langle\ \#_l\ \mathbf{cap}(\overrightarrow{v}) \mid \mathcal{F} :: E \hookrightarrow K\ \rangle : \tau$

(2) $m' = \langle\ \sigma_{sp} \mid \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) :: E\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \#_l\ \mathbf{cap}(\overrightarrow{v}) : \tau_1$

(4) $\vdash_{ctx}\ \mathcal{F} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi$

(5) $\Xi \vdash_{cnt}\ K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

By inversion (4)

(6) $\vdash_{ctx}\ E : \tau_{rt} \rightsquigarrow \tau \mid \Xi$

(7) $\Xi \vdash_{\mathcal{F}}\ \mathcal{F} : \tau_1 \rightsquigarrow \tau_{rt} \mid \tau_{sp}$

By inversion (7)

(8) $\bullet \mid \bullet \mid \Xi \vdash \ s_{sp} : \tau_{sp}$

(9) $x : \tau_{sp} \mid \bullet \mid \Xi \vdash \ s_{rs} : \mathsf{Unit}$

(10) $x : \tau_2 \mid \bullet \mid \Xi \vdash \ s_{rt} : \tau_{rt}$

By inversion (3)

(11) $l : \overrightarrow{\tau_i} \to \tau_1 \in \Xi$

(12) $\overrightarrow{\bullet \vdash \ v : \tau_i}$

By [F-Unwind] applied to (5), (7), (11) and (12)

(13) $\Xi \vdash_{frm}\ \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) : \tau_{sp} \to \tau_{rt}$

Applying [E-Frame] to (6) and (13) yields

(14) $\vdash_{ctx}\ \#_l\ \mathbf{unwind}(\overrightarrow{v},\ \mathcal{F},\ K) :: E : \tau_{sp} \rightsquigarrow \tau \mid \Xi$

Finally, applying [M-Reduction] to (8) and (14) gives us the desired result.

Case M-Rewind:
Sub-Case resume:
We know
(1) $m = \langle\ \mathbf{return}\ v \mid E \hookleftarrow\!\!\rho\ (\mathcal{F},\ v')\ ::\ K\ \rangle$
(2) $m' = \langle\ s_{rs} \mid \#\,\mathbf{rewind}(v,\ \mathcal{F},\ K)\ ::\ E\ \rangle$
By inversion on (1)
(3) $\bullet \mid \bullet \mid \Xi \vdash\ \mathbf{return}\ v : \tau_1$
(4) $\vdash_{ctx}\ E : \tau_{rt} \rightsquigarrow \tau \mid \Xi$
(5) $\Xi \vdash_{cnt}\ (\mathcal{F},\ v)\ ::\ K : \tau_1 \rightsquigarrow \tau_{rt} \mid \Xi'$
By inversion on (6)
(6) $\Xi \vdash_{cnt}\ K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$
(7) $\Xi \vdash_{\mathcal{F}}\ \mathcal{F} : \tau_2 \rightsquigarrow \tau_{rt} \mid \tau_{sp}$
(8) $\bullet \vdash\ v : \tau_{sp}$
By inversion on (7)
(9) $\bullet \mid \bullet \mid \Xi \vdash\ s_{sp} : \tau_{sp}$
(10) $x : \tau_{sp} \mid \bullet \mid \Xi \vdash\ s_{rs} : \mathsf{Unit}$
(11) $x : \tau_2 \mid \bullet \mid \Xi \vdash\ s_{rt} : \tau_{rt}$
By applying [F-Rewind] to (6), (7) and (8), we get
(12) $\Xi \vdash_{frm}\ \#\,\mathbf{rewind}(v,\ \mathcal{F},\ K) : \mathsf{Unit} \rightsquigarrow \tau_{rt}$
By applying [E-Frame] to (4) and (12), we get
(13) $\vdash_{ctx}\ \#\,\mathbf{rewind}(v,\ \mathcal{F},\ K)\ ::\ E : \mathsf{Unit} \rightsquigarrow \tau \mid \Xi$
By using Lemma [#lm-subst-value] with (8) and (10), we learn
(14) $\bullet \mid \bullet \mid \Xi \vdash\ s_{rs}[x \mapsto v] : \mathsf{Unit}$
Finally, by using [M-Reduction] with (14) and (13) we get the desired result.
Sub-Case rewind:
We know
(1) $m = \langle\ \mathbf{return}\ v \mid E \hookleftarrow\!\!\rho\ F\ ::\ K\ \rangle : \tau$
(2) $m' = \langle\ \mathbf{return}\ v \mid F\ ::\ E \hookleftarrow\!\!\rho\ K\ \rangle$
By inversion on (1)
(3) $\bullet \mid \bullet \mid \Xi \vdash\ \mathbf{return}\ v : \tau_1$
(4) $\Xi \vdash_{cnt}\ F\ ::\ K\ :\ \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$
(5) $\vdash_{ctx}\ E : \tau_2 \rightsquigarrow \tau_3 \mid \Xi$
By inversion on (4)
(6) $\Xi \vdash_{frm}\ F : \tau_2' \rightsquigarrow \tau_2$
(7) $\Xi \vdash_{cnt}\ K\ :\ \tau_1 \rightsquigarrow \tau_2' \mid \Xi'$
By applying [E-Frame] on (5) and (6)
(8) $\vdash_{ctx}\ F\ ::\ E\ :\ \tau_2' \rightsquigarrow \tau_3 \mid \Xi$
Apply M-Rewind to (3), (7) and (8).
Sub-Case rewind':
We know
(1) $\langle\ \mathbf{return}\ v \mid E \hookleftarrow\!\!\rho\ \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ K\ \rangle : \tau$
(2) $\langle\ \mathbf{return}\ v \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\ ::\ E \hookleftarrow\!\!\rho\ K\ \rangle$
By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \mathbf{return}\ v : \tau_1$
(4) $\vdash_{ctx} E : \tau_2 \rightsquigarrow \tau \mid \Xi$
(5) $\Xi \vdash_{cnt} \#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x},\ k) \Rightarrow s \} :: K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

By inversion on (5)

(6) $\Xi,\ l : \overrightarrow{\tau_i} \to \tau_0 \vdash_{cnt} K : \tau_1 \rightsquigarrow \tau_2 \mid \Xi \setminus l$
(7) $\overrightarrow{x : \tau_i} \mid k : \tau_0 \to \tau_2 \mid \Xi \vdash \ s : \tau_2$

By applying [E-Handler] to (4) and (7)

(8) $\vdash_{ctx} \#_l \{ \square \}\ \mathbf{with}\ \{ (\overrightarrow{x},\ k) \Rightarrow s \} :: E : \tau_2 \rightsquigarrow \tau \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0$

We get the desired result by applying M-Rewind to (3), (6) and (8).

Sub-Case stop:

We know

(1) $m = \langle\ \mathbf{return}\ v \mid E \leftharpoonup \bullet\ \rangle : \tau$
(2) $m' = \langle\ \mathbf{return}\ v \mid E\ \rangle$

By inversion on (1)

(3) $\bullet \mid \bullet \mid \Xi \vdash \ \mathbf{return}\ v : \tau_1$
(4) $\vdash_{ctx} E : \tau_2 \rightsquigarrow \tau \mid \Xi$
(5) $\Xi \vdash \ \bullet : \tau_1 \rightsquigarrow \tau_2 \mid \Xi'$

By inversion on (5), we learn

(6) $\tau_1 = \tau_2$

Applying M-Reduction to (3) and (4) with (6) yields the desired result. □

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