Dynamic Wind for Effect Handlers

OOPSLA @ ICFP/SPLASH 2025 – 17. October 2025

David Voigt Philipp Schuster Jonathan Brachthäuser

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Funded by
DFG Deutsche Forschungsgemeinschaft
German Research Foundation

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.

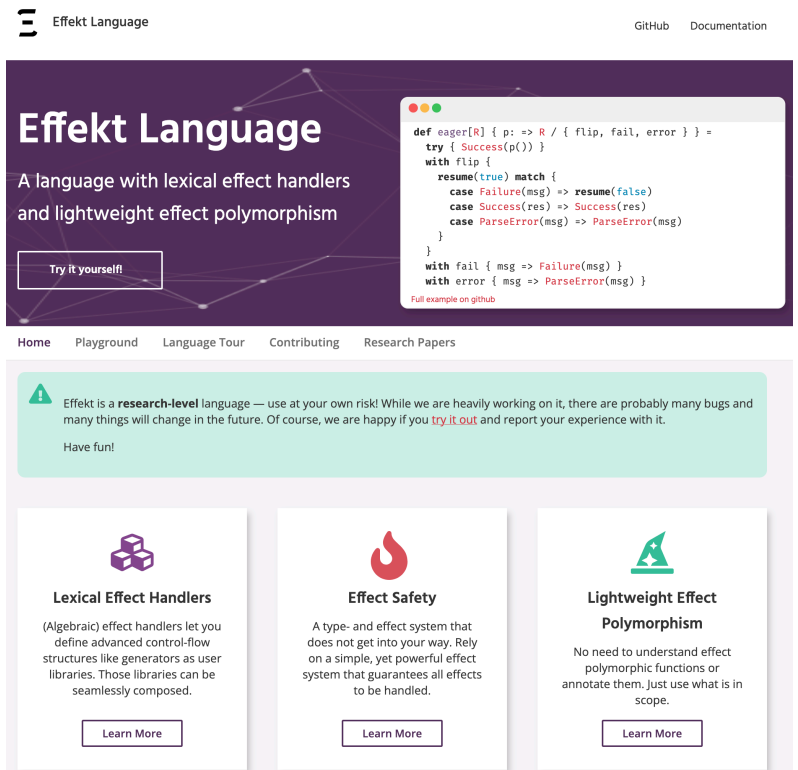
CCS Concepts: • **Software and its engineering** → **Control structures**; **Compilers**; • **Theory of computation** → *Type theory*; **Control primitives**.

Context

- **Lexical** Effect Handlers
  - **Multi-Shot continuations**
  - Complex user-controlled control flow: async, non-determinism, probabilistic programming, ...

Context

- **Lexical** Effect Handlers
  - **Multi-Shot continuations**
  - Complex user-controlled control flow: async, non-determinism, probabilistic programming, …
- Practical implementation as a fork of the **Effekt** compiler



https://effekt-lang.org

Example

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  if (do flip()) {
    msg.set(msg.get ++ ", world"!)
  } else {
    msg.set(msg.get ++ ", singapore!")
  }
  println(msg.get)
}
```

Example

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  if (do flip()) {
    msg.set(msg.get ++ ", world"!)
  } else {
    msg.set(msg.get ++ ", singapore!")
  }
  println(msg.get)
}
```

```
def handleFlip[A] { prog: ⇒ A }: A =
  try prog()
  with flip {
    resume(true)
    resume(false)
  }
```

Example

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  if (do flip()) {
    msg.set(msg.get ++ ", world"!)
  } else {
    msg.set(msg.get ++ ", singapore!")
  }
  println(msg.get)
}
```

```
def main() = {
  with handleFlip
  var msg = "hello"
  if (do flip()) {
    msg = "hello" ++ ", world!"
  } else {
    msg = "hello" ++ ", singapore!"
  }
  println(msg)
}
```

```
hello, world!
hello, singapore!
```

Example

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  if (do flip()) {
    msg.set(msg.get ++ ", world"!)
  } else {
    msg.set(msg.get ++ ", singapore!")
  }
  println(msg.get)
}
```

```
hello, world!
hello, world!, singapore!
```

Problem

- Allow **backtracking** or **finalization** of external or global resources
  - E.g. database connections, file handles, global mutable state

Problem

- Allow **backtracking** or **finalization** of external or global resources
  - E.g. database connections, file handles, global mutable state
- Even in the presence of **complex control-flow** (multi-shot delimited continuations)

Problem

- Allow **backtracking** or **finalization** of external or global resources
  - E.g. database connections, file handles, global mutable state
- Even in the presence of **complex control-flow** (multi-shot delimited continuations)

Solution

- Make the process of **suspending** and **resuming** a computation observable
  - `on suspend` runs when suspending
  - `on resume` runs when resuming

```
try { s }
on suspend { s }
on resume { x ⇒ s }
```

Example – Resumed

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  with backtrack(msg)
  if (do flip()) {
    msg.set(msg.get ++ ", world")
  } else {
    msg.set(msg.get ++ ", singapore")
  }
  println(msg.get)
}
```

```
def backtrack[A](
  r: Ref[String]
) { prog: ⇒ A }: A =
  ...
```

Example – Resumed

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  with backtrack(msg)
  if (do flip()) {
    msg.set(msg.get ++ ", world")
  } else {
    msg.set(msg.get ++ ", singapore")
  }
  println(msg.get)
}
```

```
def backtrack[A](
  r: Ref[String]
) { prog: ⇒ A }: A =
  try prog()
  on suspend { ... }
  on resume { ... }
```

Example – Resumed

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  with backtrack(msg)
  if (do flip()) {
    msg.set(msg.get ++ ", world")
  } else {
    msg.set(msg.get ++ ", singapore")
  }
  println(msg.get)
}
```

```
def backtrack[A](
  r: Ref[String]
) { prog: ⇒ A }: A =
  try prog()
  on suspend { r.get }
  on resume { ... }
```

Example – Resumed

```
def main() = {
  with handleFlip
  val msg = ref("hello")
  with backtrack(msg)
  if (do flip()) {
    msg.set(msg.get ++ ", world")
  } else {
    msg.set(msg.get ++ ", singapore")
  }
  println(msg.get)
}
```

```
def backtrack[A](
  r: Ref[String]
) { prog: ⇒ A }: A =
  try prog()
  on suspend { r.get }
  on resume { s => r.set(s) }
```

Example – Resumed

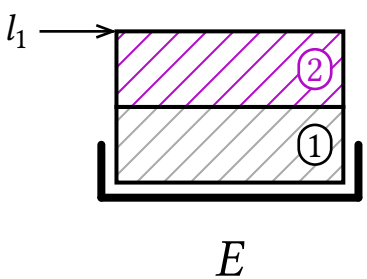
```
def main() = {
  with handleFlip
  val msg = ref("hello")
  with backtrack(msg)
  if (do flip()) {
    msg.set(msg.get ++ ", world")
  } else {
    msg.set(msg.get ++ ", singapore")
  }
  println(msg.get)
}
```

```
def backtrack[A](
  r: Ref[String]
) { prog: ⇒ A }: A =
  try prog()
  on suspend { r.get }
  on resume { s ⇒ r.set(s) }
```

```
hello, world!
hello, singapore!
```

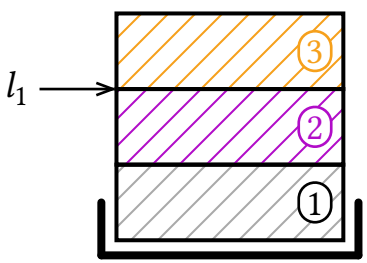
How does it work semantically?

Suspending



② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

Suspending

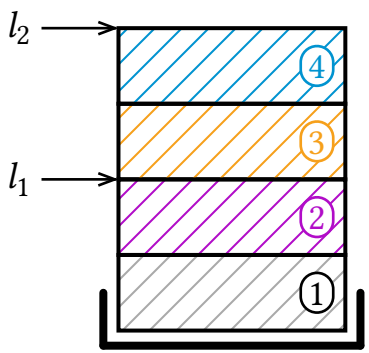


E

② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

Suspending



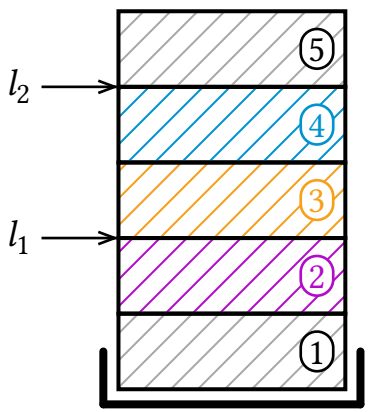
E

② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending

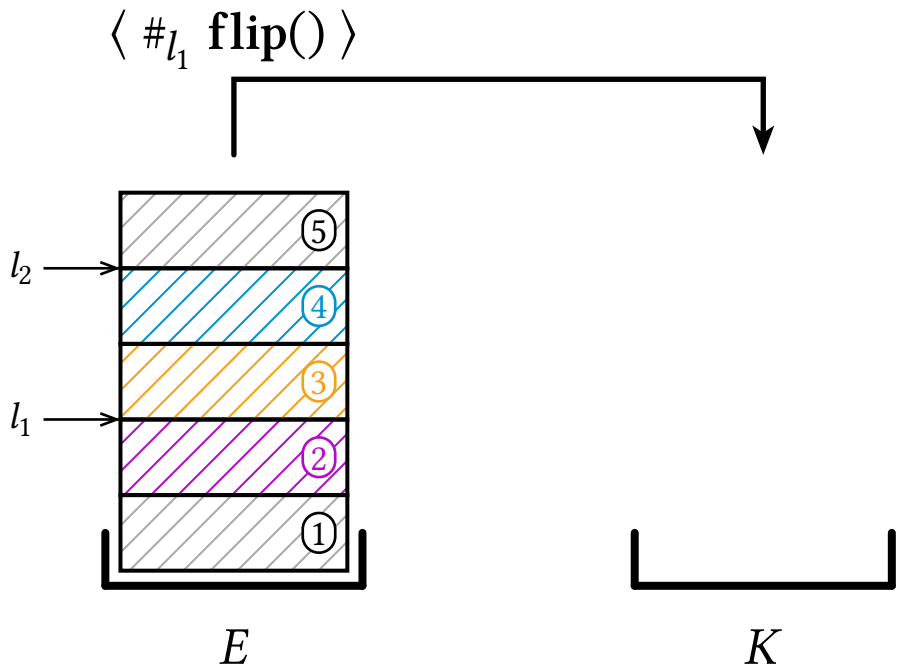


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending

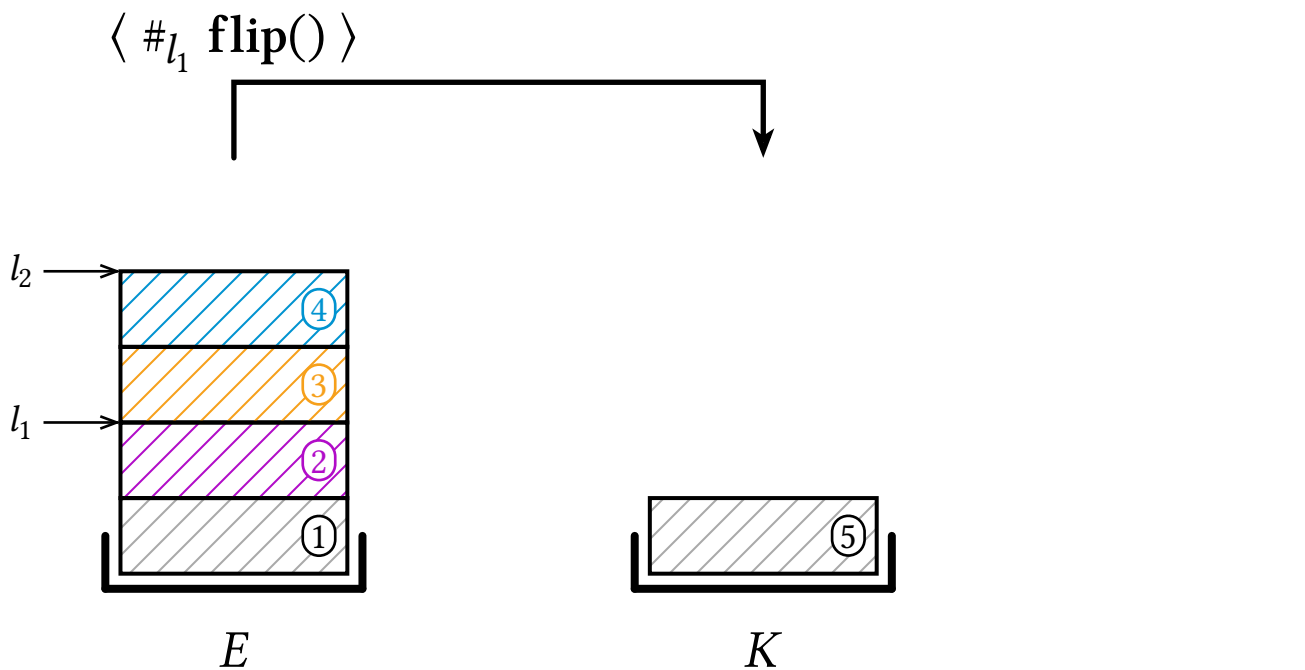


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending

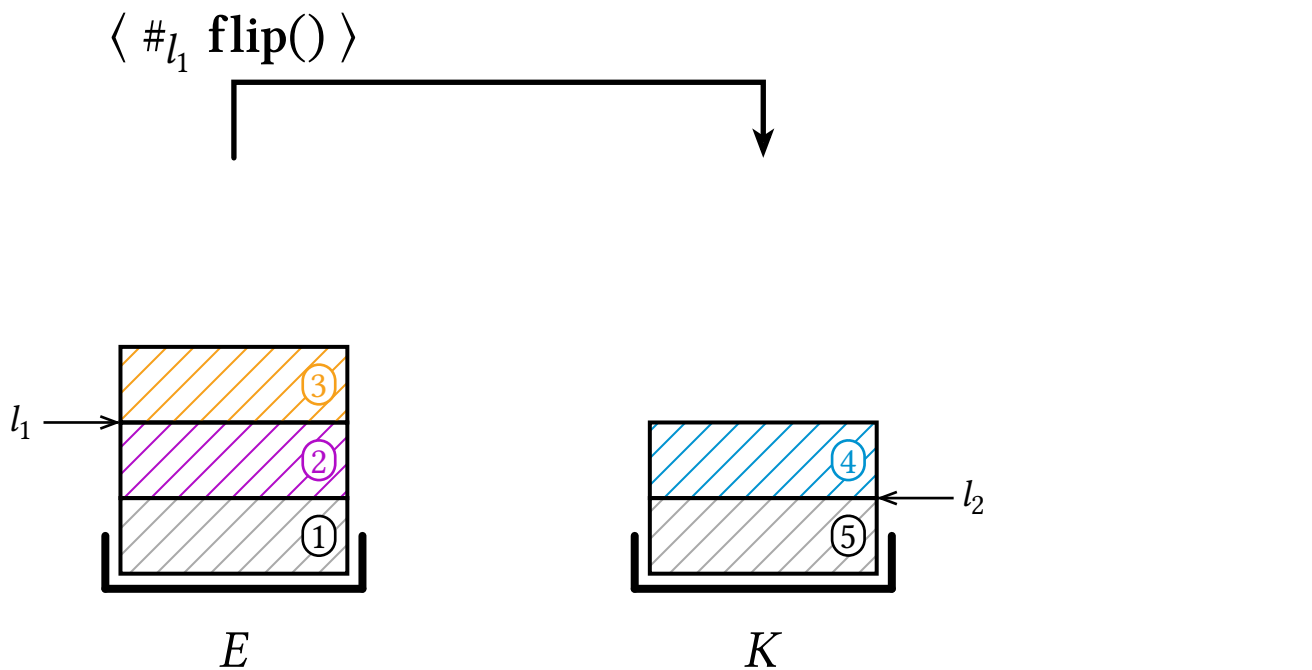


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending

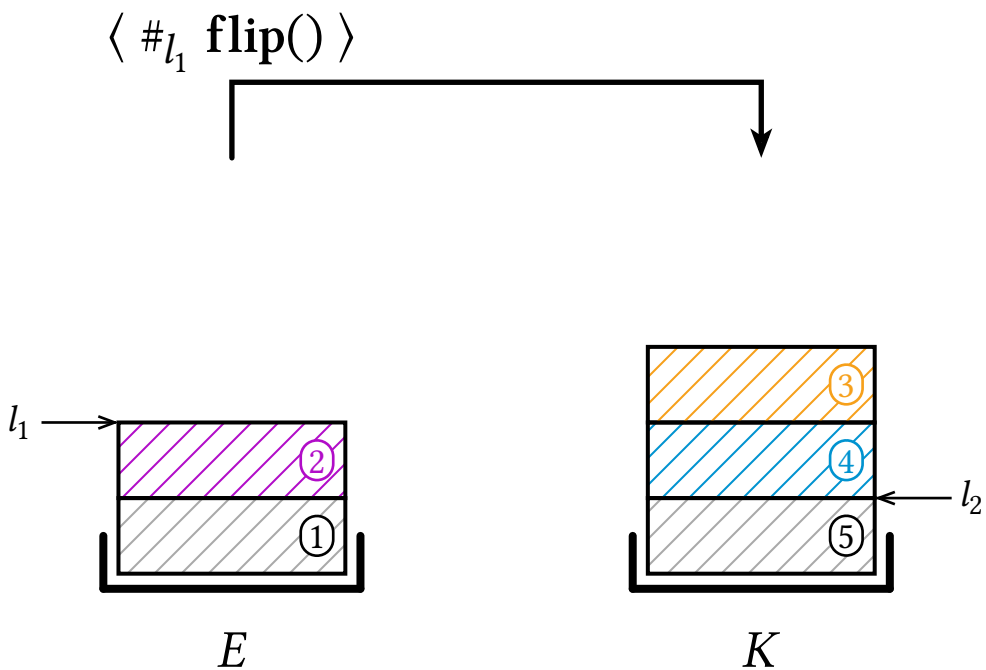


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending

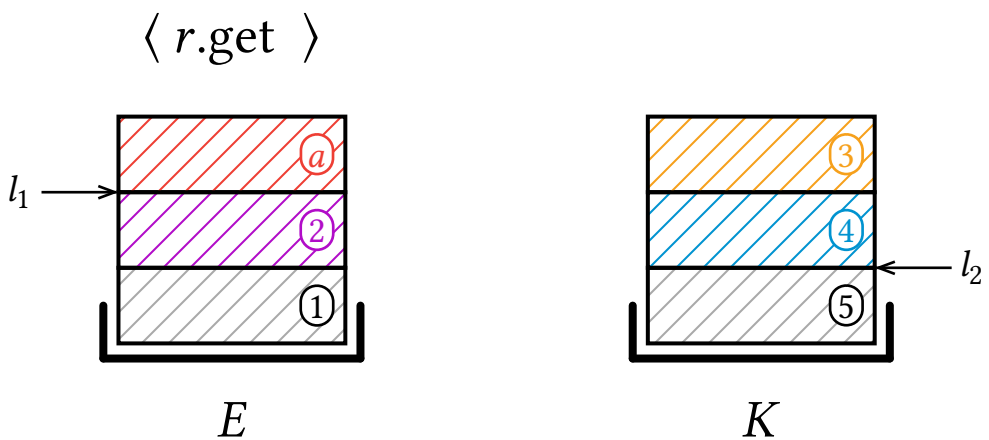


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Suspending



② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

ⓐ := $\#_{l_1}$ **unwind**((), K)

Suspending

⟨ **return** "hello" ⟩

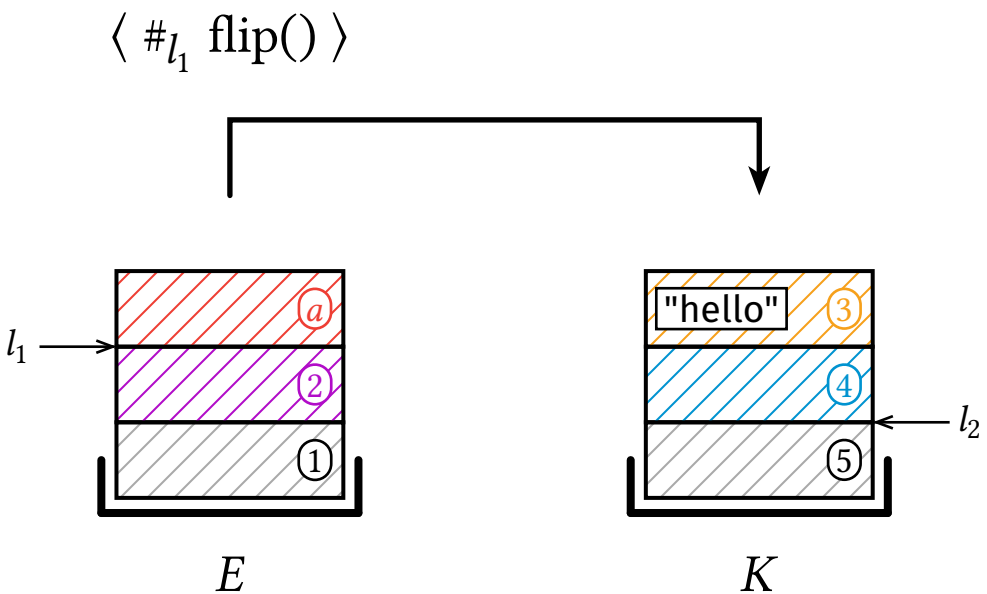
② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

ⓐ := $\#_{l_1}$ **unwind**((), K)

Suspending



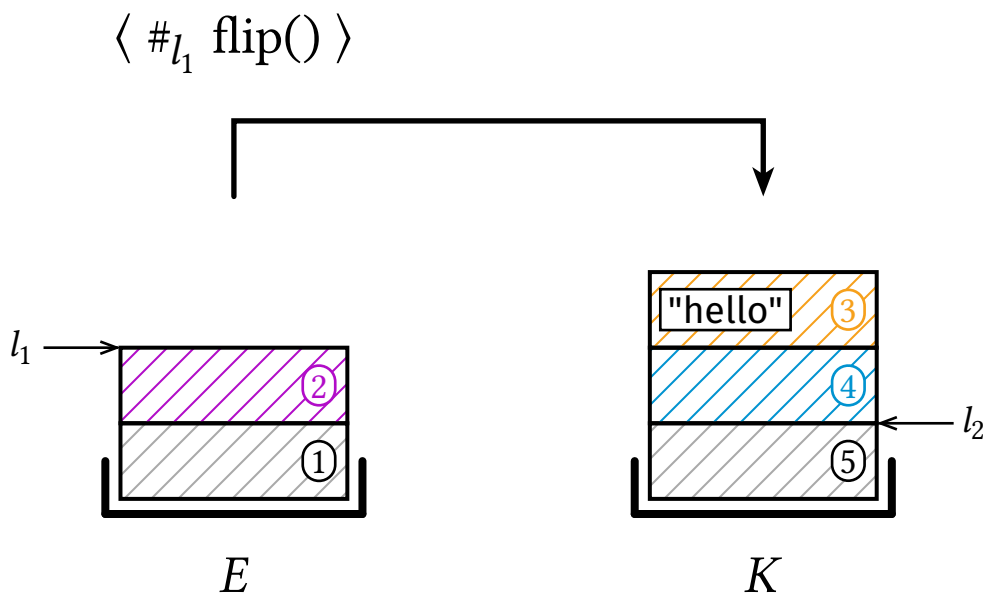
② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

ⓐ := #$_{l_1}$ **unwind**((), K)

Suspending



② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

What about the other direction? What about resuming?

Resuming

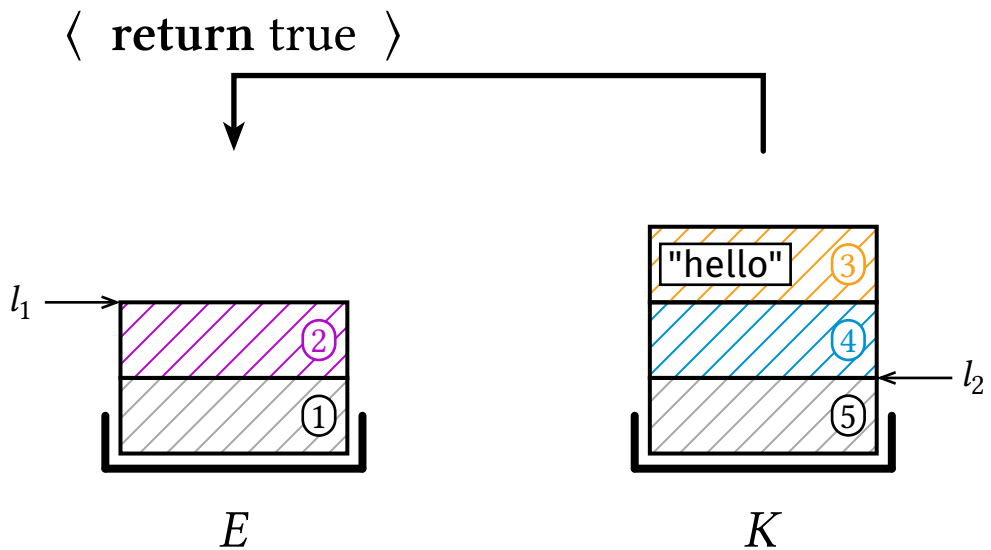
$\langle$ **return** true $\rangle$

② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming



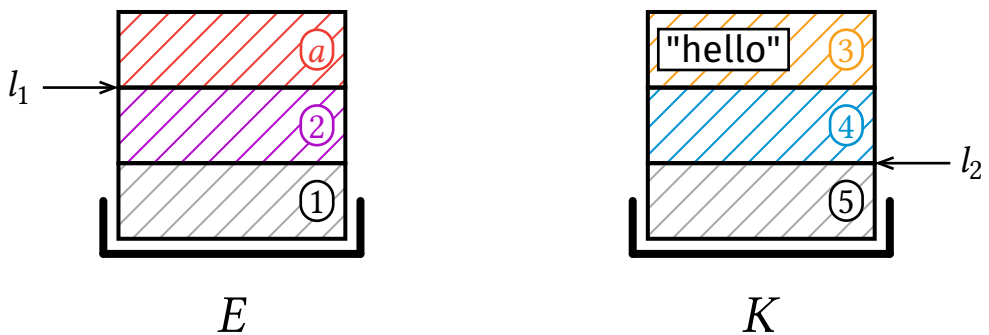
② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming

$\langle\ r.\text{set}(\text{"hello"})\ \rangle$



② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

ⓐ := # **rewind**(true, K)

Resuming

$\langle$ return () $\rangle$

② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

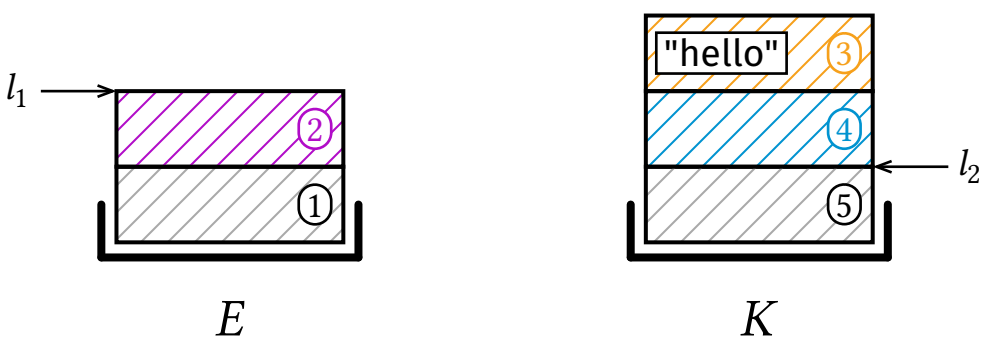
③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

ⓐ := # **rewind**(true, K)

Resuming

⟨ **return** true ⟩

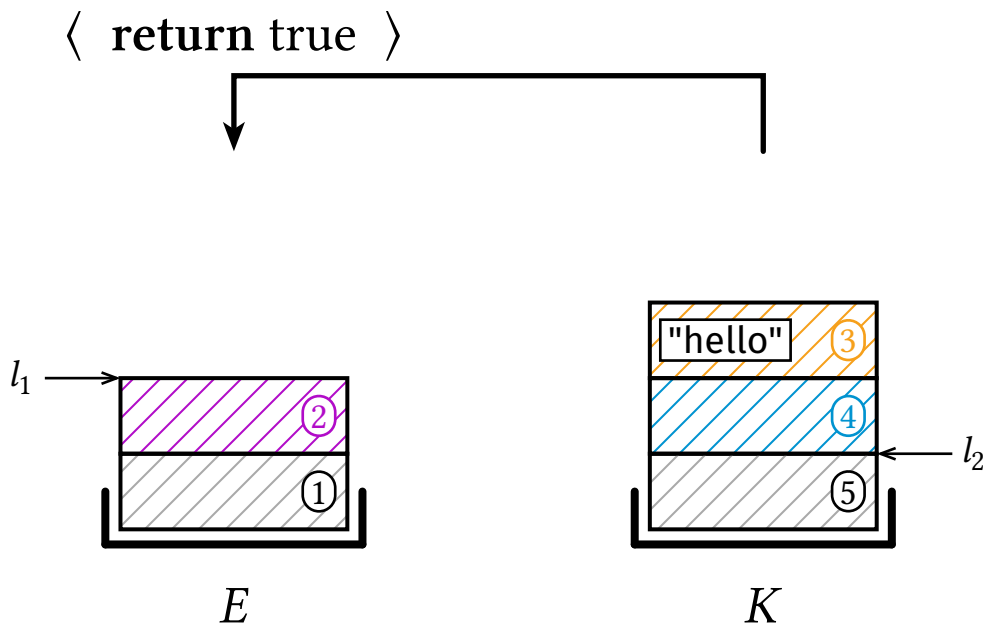


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming

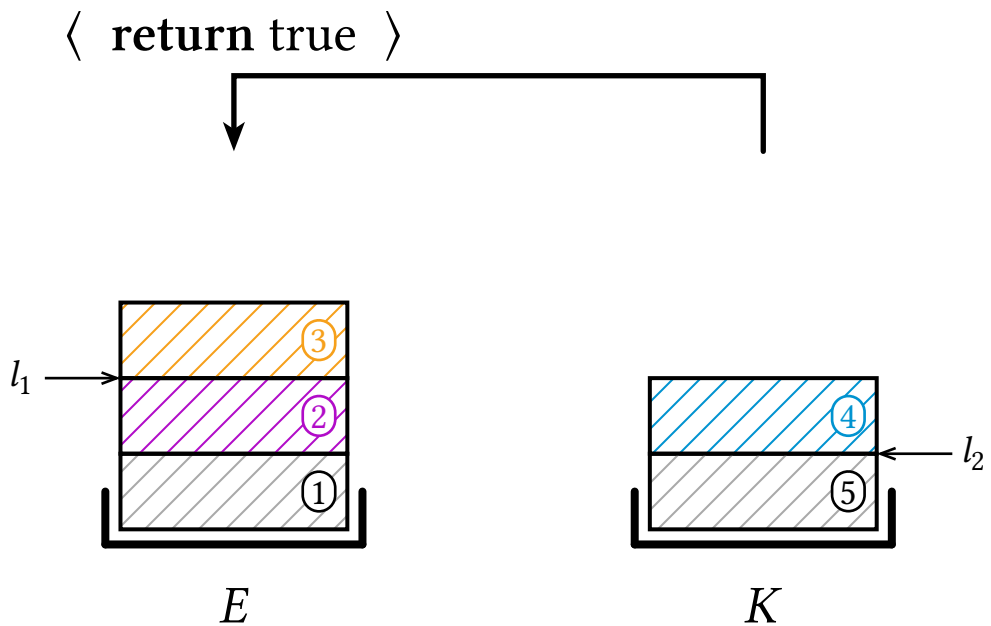


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming

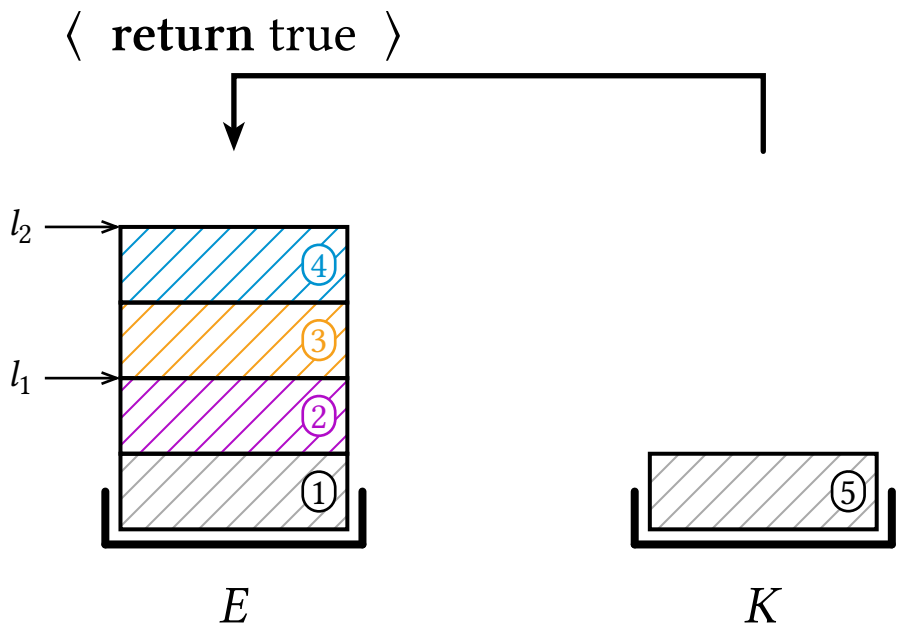


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming

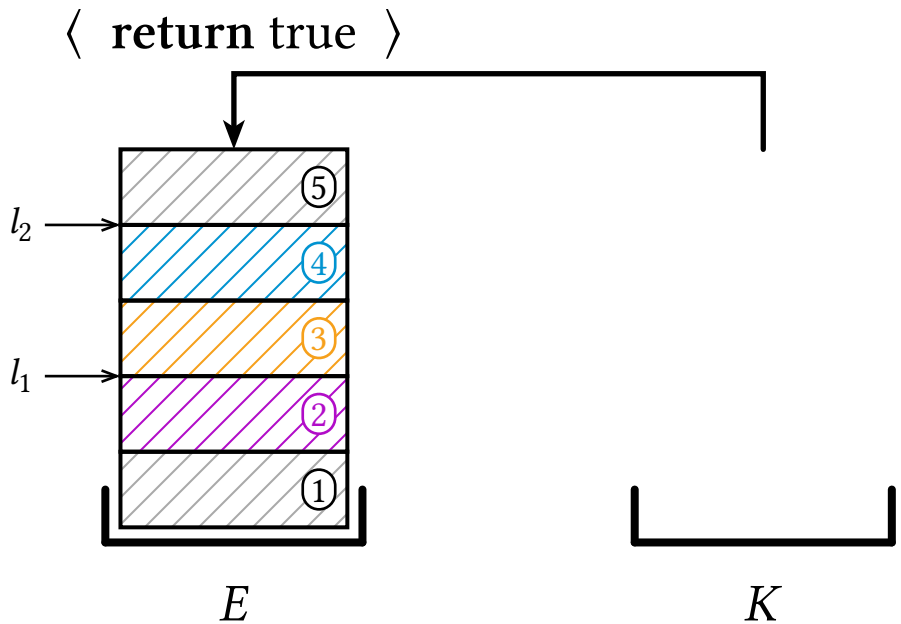


② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming



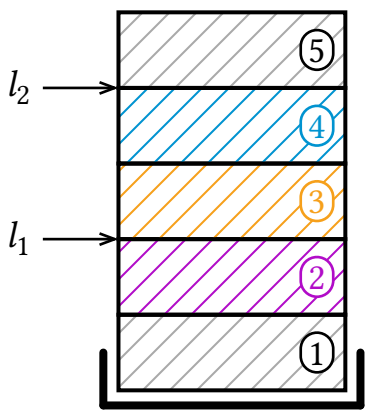
② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Resuming

$\langle$ **return** true $\rangle$



E

② := **try** { □ } **with** flip { _ ⇒ **resume**(true); **resume**(false) }

③ := { □ } **on suspend** { r.get } **on resume** { x ⇒ r.set(x) }

④ := **try** { □ } **with** ◇

Theoretical Results

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

Theoretical Results

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

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

Theoretical Results

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

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

Theorem (Soundness). If $\vdash_m M : t$, then there exists a step $M \longrightarrow^* M$ and there exists no further step $M' \longrightarrow M''$, then $\vdash_m M' : \tau$ and such that $M' = \langle\, \text{return}\ v \mid \bullet \,\rangle$.

Theoretical Results

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

Theoretical Results

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

```
val file = ref(open(path))
try { ... read(file) ... }
on suspend { close(file); ... }
on resume { ...; file.set(open(path)) }
on return { x ⇒ close(file); ... }
```

Theoretical Results

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

```
val file = ref(open(path))
try { ... read(file) ... }
on suspend { close(file); ... }
on resume { ...; file.set(open(path)) }
on return { x ⇒ close(file); ... }
```

- All resources on the stack are open / available
- All other resources – potentially captured in continuations – are closed / finalized

Summary

- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers

Summary

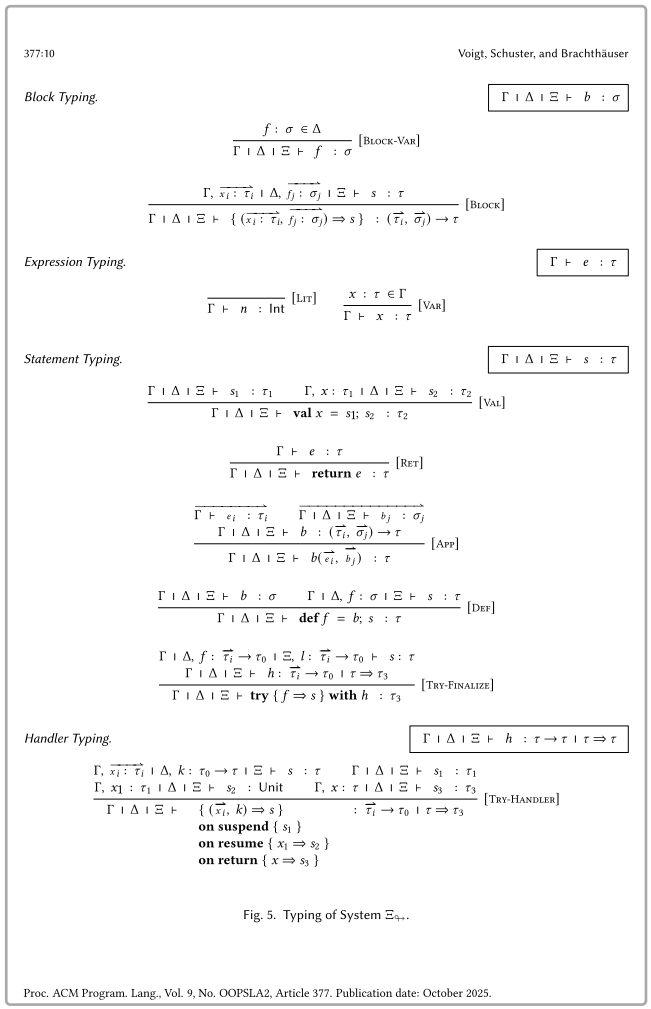
- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers
- We added new finalization clauses: `on suspend`, `on resume` and `on return`
- Notably, calling effect operations is **well-behaved** in the finalization clauses

Summary

- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers
- We added new finalization clauses: `on suspend`, `on resume` and `on return`
- Notably, calling effect operations is **well-behaved** in the finalization clauses

In the Paper

- Full formalization of $\Xi_{\leftrightarrow}$ including **operational semantics** and a declarative **type system**



377:10 Voigt, Schuster, and Brachthäuser

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

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

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

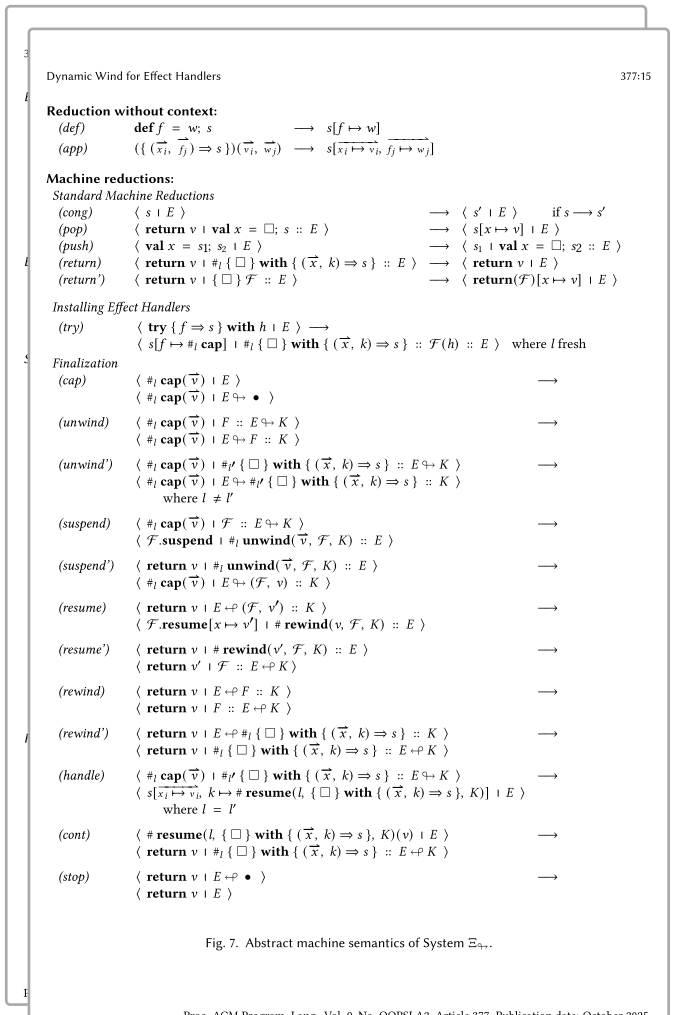
Proc. ACM Program. Lang., Vol. 9, No. OOPSLA2, Article 377. Publication date: October 2025.

Summary

- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers
- We added new finalization clauses: `on suspend`, `on resume` and `on return`
- Notably, calling effect operations is **well-behaved** in the finalization clauses

In the Paper

- Full formalization of $\Xi_{\hookrightarrow}$ including **operational semantics** and a declarative **type system**



Dynamic Wind for Effect Handlers 377:15

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

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

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

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

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

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

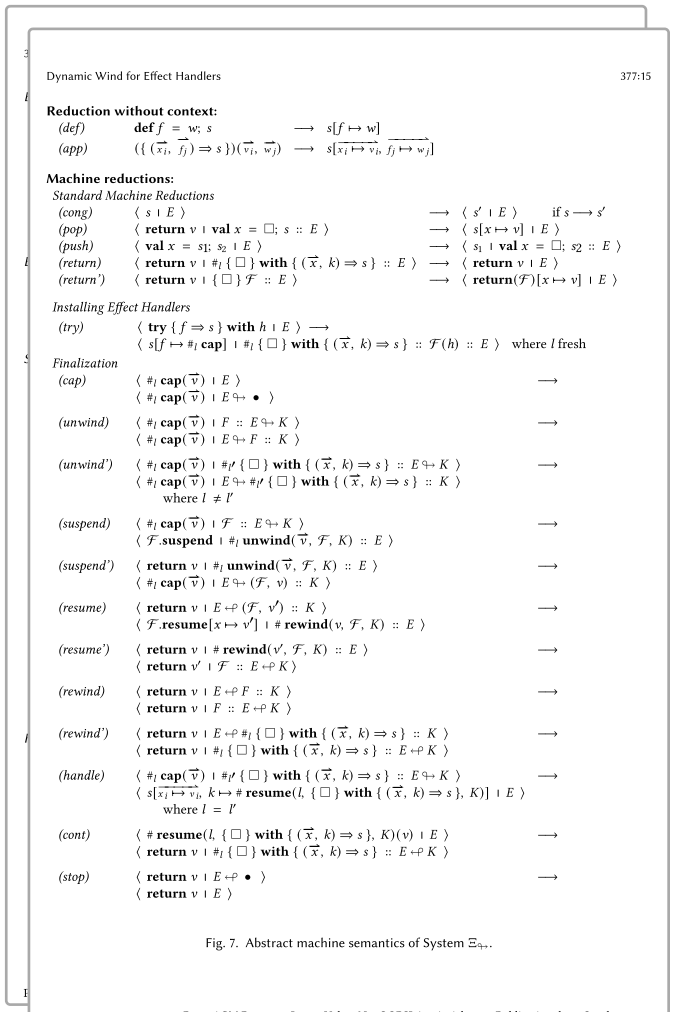
Proc. ACM Program. Lang., Vol. 9, No. OOPSLA2, Article 377. Publication date: October 2025.

Summary

- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers
- We added new finalization clauses: `on suspend`, `on resume` and `on return`
- Notably, calling effect operations is **well-behaved** in the finalization clauses

In the Paper

- Full formalization of $\Xi_{\hookrightarrow}$ including **operational semantics** and a declarative **type system**
- Progress and Preservation theorems & proofs
- Resource Safety theorem & proof



Dynamic Wind for Effect Handlers 377:15

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

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

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

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

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

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

Proc. ACM Program. Lang., Vol. 9, No. OOPSLA2, Article 377. Publication date: October 2025.

Summary

- **Multi-shot delimited continuations** require special care when adding **dynamic finalization** to (lexical) effect handlers
- We added new finalization clauses: `on suspend`, `on resume` and `on return`
- Notably, calling effect operations is **well-behaved** in the finalization clauses

In the Paper

- Full formalization of $\Xi_{\leftrightarrow}$ including **operational semantics** and a declarative **type system**
- Progress and Preservation theorems & proofs
- Resource Safety theorem & proof

Link to the complete paper

Effect Parametricity

Parametricity can be understood as interpreting a syntactic universal type as meta-level universal quantification over a given universe of semantic types (Vilhena and Pottier 2023).

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

- In Effekt: no means of universally abstracting over effects due to *Contextual Effect Polymorphism*
- `catchAll` behaves the same for all possibly occuring effects
- We leave it to future work what Effect Parametricity means w.r.t. Contextual Effect Polymorphism compared to parametric quantification over effects

Tail-Resumption Optimization

- Optimization for removing continuation capturing if the handler removes exactly once in tail position:

```
try {
  try { log("hello") }
  on suspend {
    println("suspending")
  } on resume { _ ⇒
    println("resuming")
  }
} with log { msg ⇒
  println(msg)
}
```

```
def log(msg: String): Unit =
println(msg)
try { log("hello") }
on suspend {
  println("suspending")
} on resume { _ ⇒
  println("resuming")
}
```

- Replaces effect operation call with normal function call – no explicit unwinding and rewinding
- `on suspend` and `on resume` are *not* executed – non-semantics preserving!

Bibliography

[1] P. E. de Vilhena and F. Pottier, “A Type System for Effect Handlers and Dynamic Labels,” in *Programming Languages and Systems*, T. Wies, Ed., Cham: Springer Nature Switzerland, 2023, pp. 225–252. doi: 10.1007/978-3-031-30044-8_9.