

Lecture 26 – Effect Systems

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Substructural Types
- The **structural rules** – **Exchange**, **Weakening**, **Contraction** – make a variable a **fact**
- **Dropping** them yields a spectrum: affine, relevant, **linear**, ordered
- A **linear** λ -calculus tags types $\text{lin } P \mid \text{un } P$; **splitting** the context + **no weakening** \Rightarrow used **exactly once**
- **Rust** ownership is **affine**: one owner; moving invalidates the old name

Today: **effect systems** – typing not just **what a value is**, but **what a computation does**.

1. Why Effect Systems?
2. Type-and-Effect Systems
3. Algebraic Effects & Handlers
Examples by Resume Count
4. Effect Systems in Practice

1. Why Effect Systems?
2. Type-and-Effect Systems
3. Algebraic Effects & Handlers
Examples by Resume Count
4. Effect Systems in Practice

An ordinary type gives the **shape** of the result – never the **side effects** along the way. **Koka**¹ puts them **in the type**:

```
fun square( x : int ) : total int           // `total` = pure, no effect
  x * x

fun risky( x : int ) : <exn,console> int    // may throw AND print
  if x < 0 then throw("neg") else { println(x); x }
```

¹<https://koka-lang.github.io/>

An ordinary type gives the **shape** of the result – never the **side effects** along the way. **Koka**¹ puts them **in the type**:

```
fun square( x : int ) : total int           // `total` = pure, no effect
  x * x

fun risky( x : int ) : <exn,console> int    // may throw AND print
  if x < 0 then throw("neg") else { println(x); x }
```

The primitive effects we track:

exn (throw) io (I/O) st (state) div (non-termination)

¹<https://koka-lang.github.io/>

An ordinary type gives the **shape** of the result – never the **side effects** along the way. **Koka**¹ puts them **in the type**:

```
fun square( x : int ) : total int           // `total` = pure, no effect
  x * x

fun risky( x : int ) : <exn,console> int    // may throw AND print
  if x < 0 then throw("neg") else { println(x); x }
```

The primitive effects we track:

exn (throw) io (I/O) st (state) div (non-termination)

So, the judgment becomes **type-and-effect**:

$$\Gamma \vdash e : \tau ! \varepsilon$$

which means that “*e* has type τ **with effect** ε ”.

¹<https://koka-lang.github.io/>

1. Why Effect Systems?
2. Type-and-Effect Systems
3. Algebraic Effects & Handlers
Examples by Resume Count
4. Effect Systems in Practice

An Effect Is a Set of Things That May Happen

An **effect** ε is a **set** of primitive labels; the empty set \emptyset means **pure**.

Effects **combine by union** and are ordered by **inclusion**:

$$\emptyset \subseteq \{\text{exn}\} \subseteq \{\text{exn}, \text{io}\} \subseteq \{\text{exn}, \text{io}, \text{st}\} \subseteq \dots$$

An **effect** ε is a **set** of primitive labels; the empty set \emptyset means **pure**.
Effects **combine by union** and are ordered by **inclusion**:

$$\emptyset \subseteq \{\text{exn}\} \subseteq \{\text{exn}, \text{io}\} \subseteq \{\text{exn}, \text{io}, \text{st}\} \subseteq \dots$$

So $(\mathcal{P}(\{\text{exn}, \text{io}, \text{st}, \text{div}\}), \subseteq, \cup)$ is a **join semilattice** – exactly the structure we need to **accumulate** effects.

A function **definition** does nothing; the effect happens **when it is called**.

So, the effect is **latent** on the arrow:

$$\tau_1 \xrightarrow{\varepsilon} \tau_2 \quad (\text{Koka writes this } \text{int} \rightarrow \langle \text{exn} \rangle \text{ int})$$

A function **definition** does nothing; the effect happens **when it is called**.

So, the effect is **latent** on the arrow:

$$\tau_1 \xrightarrow{\varepsilon} \tau_2 \quad (\text{Koka writes this } \text{int} \rightarrow \langle \text{exn} \rangle \text{ int})$$

Types $\tau ::= \text{bool} \mid \text{int} \mid \tau_1 \xrightarrow{\varepsilon} \tau_2$

Effects $\varepsilon ::= \text{a set } \subseteq \{\text{exn}, \text{io}, \text{st}, \text{div}\}$

Terms $e ::= x \mid \lambda x:\tau. e \mid e e \mid \text{throw } e \mid \text{try } e \text{ catch } e$

Typing Rules: Values Are Pure, Application Unions

Values produce **no** effect; T-ABS **discharges** the body's effect ε **onto the arrow**:

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau ! \emptyset} \text{T-VAR}$$

$$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2 ! \varepsilon}{\Gamma \vdash \lambda x:\tau_1. e : (\tau_1 \xrightarrow{\varepsilon} \tau_2) ! \emptyset} \text{T-ABS}$$

Values produce **no** effect; T-ABS **discharges** the body's effect ε **onto the arrow**:

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau ! \emptyset} \text{T-VAR} \qquad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2 ! \varepsilon}{\Gamma \vdash \lambda x:\tau_1. e : (\tau_1 \xrightarrow{\varepsilon} \tau_2) ! \emptyset} \text{T-ABS}$$

Application unions **three effects** – 1) the function, 2) the argument, and 3) the latent effect ε released at the call site:

$$\frac{\Gamma \vdash e_1 : (\tau_1 \xrightarrow{\varepsilon} \tau_2) ! \varepsilon_1 \quad \Gamma \vdash e_2 : \tau_1 ! \varepsilon_2}{\Gamma \vdash e_1 e_2 : \tau_2 ! \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon} \text{T-APP}$$

`throw` **adds** the `exn` effect (and may have **any** result type – it never returns normally):

$$\frac{\Gamma \vdash e : \tau ! \varepsilon}{\Gamma \vdash \text{throw } e : \tau' ! \varepsilon \cup \{\text{exn}\}} \text{T-THROW}$$

throw **adds** the `exn` effect (and may have **any** result type – it never returns normally):

$$\frac{\Gamma \vdash e : \tau ! \varepsilon}{\Gamma \vdash \text{throw } e : \tau' ! \varepsilon \cup \{\text{exn}\}} \text{T-THROW}$$

A **handler** is the **only** way to **remove** an effect: try **subtracts** `exn` from the protected expression.

$$\frac{\Gamma \vdash e_1 : \tau ! \varepsilon_1 \quad \Gamma \vdash e_2 : \tau ! \varepsilon_2}{\Gamma \vdash \text{try } e_1 \text{ catch } e_2 : \tau ! (\varepsilon_1 \setminus \{\text{exn}\}) \cup \varepsilon_2} \text{T-TRY}$$

`throw` **adds** the `exn` effect (and may have **any** result type – it never returns normally):

$$\frac{\Gamma \vdash e : \tau ! \varepsilon}{\Gamma \vdash \text{throw } e : \tau' ! \varepsilon \cup \{\text{exn}\}} \text{T-THROW}$$

A **handler** is the **only** way to **remove** an effect: `try` **subtracts** `exn` from the protected expression.

$$\frac{\Gamma \vdash e_1 : \tau ! \varepsilon_1 \quad \Gamma \vdash e_2 : \tau ! \varepsilon_2}{\Gamma \vdash \text{try } e_1 \text{ catch } e_2 : \tau ! (\varepsilon_1 \setminus \{\text{exn}\}) \cup \varepsilon_2} \text{T-TRY}$$

Primitives introduce effects, **handlers eliminate** them. Subtracting `exn` is well-defined whether or not ε already contains it.

Claiming **more** effects than really occur is **sound**:

$$\frac{\Gamma \vdash e : \tau ! \varepsilon \quad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash e : \tau ! \varepsilon'} \text{T-SUB}$$

Claiming **more** effects than really occur is **sound**:

$$\frac{\Gamma \vdash e : \tau ! \varepsilon \quad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash e : \tau ! \varepsilon'} \text{T-SUB}$$

So, a conditional just takes the **join** of its branches.

$$\frac{\Gamma \vdash e_1 : \text{bool} ! \varepsilon_1 \quad \Gamma \vdash e_2 : \tau ! \varepsilon_2 \quad \Gamma \vdash e_3 : \tau ! \varepsilon_3}{\Gamma \vdash \text{if } e_1 \ e_2 \ e_3 : \tau ! \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \text{T-IF}$$

```

fun safe-div( a : int, b : int ) : exn int
  if b == 0 then throw("div by zero") else a / b

fun run() : int
  with catch( fn(err) { 0 } ) // handler discharges `exn`: return 0
  safe-div(10, 0)

```

throw(...)	:	int ! {exn}	(T-THROW)
if (b=0) ... (a/b)	:	int ! {exn}	(T-IF)
safe-div	:	(int, int) $\xrightarrow{\{exn\}}$ int	(T-ABS)
safe-div(10,0)	:	int ! {exn}	(T-APP)
catch(...)	:	int ! \emptyset	(T-TRY subtracts exn)

Note that Koka's built-in `/` is **total** ($x/0 = 0$, it never throws).

1. Why Effect Systems?
2. Type-and-Effect Systems
3. Algebraic Effects & Handlers
Examples by Resume Count
4. Effect Systems in Practice

In Koka, we can define user-defined **algebraic effects** and **handlers**:

```
// `effect` : declare a new effect whose name is `ask`
effect ask
  // `ctl` : declare a control operation (capture continuation) `ask`
  ctl ask() : int
// or simply `effect ctl ask() : int` for a single operation

fun run() : int
  // `with` (`handler`) : install a handler in the rest of the scope
  with handler {
    // `return` : transform the final result value (here: unchanged)
    return(x) x
    // `resume` : continue the suspended computation with a value
    ctl ask() { resume(21) }
  }
  // or simply `with ctl ask() { resume(21) }` when there is no `return`

  // ask() -- suspends, runs handler, resumes with 21, so 21 * 2 = 42
  ask() * 2
```

How Often You Resume Decides the Effect

The `resume` restarts the rest of the computation after the operation.

The `resume` restarts the rest of the computation after the operation.

How many times the handler calls `resume` represents different effects:

# resume calls	the rest of the computation ...	effect
0	is dropped – abort, jump out	exception
1	runs exactly once – the ordinary case	reader, state, output
≥ 2	re-runs – all branches, or stream items	nondeterminism, generator

If the handler **never** calls `resume`, the suspended computation is **discarded** – control jumps straight out:

```
effect ctl raise( msg : string ) : a

fun safe-div( a, b ) : raise int
  if b == 0 then raise("div by zero") else a / b

fun try-div( a, b ) : int
  with ctl raise(msg) { 0 } // NO `resume` -> drop the rest, return 0
  safe-div(a, b)           // try-div(10, 0) => 0
```

Resuming **exactly once** with a chosen value is the ordinary case.

Two operations over a mutable cell give **state**:

```
effect state
  ctl get() : int
  ctl put( x : int ) : ()

fun with_state( init, action )
  var s := init
  with handler
    ctl get() resume(s)           // resume with the current s
    ctl put(x) { s := x; resume(()) } // update s, then resume with ()
  action()

fun tick() : state int
  val n = get(); put(n + 1); n

fun run() : int
  with_state(10) { tick() + tick() } // 10 + 11 = 21
```

A handler may call `resume` **more than once** for a **single operation** – forking the computation to explore **every** outcome:

```
effect ctl flip() : bool

fun coins() : flip int
  (if flip() then 1 else 0) + (if flip() then 1 else 0)

fun amb(action) : list<int>
  with handler
    return(x : int) [x] // each finished run produces a singleton list
    ctl flip() resume(True) ++ resume(False) // resume BOTH ways
  action()

fun run()
  amb(coins) // [2,1,1,0] (all four outcomes)
```

where `return` converts the final result of the handled computation into another type – here, an integer x into a singleton list $[x]$.

A **generator** resumes **once per item**, so over the whole run the continuation advances **many** times – streaming values out lazily:

```
effect ctl yield( x : int ) : ()

fun upto( i : int, n : int ) : <div,yield> () // recursion adds `div`
  if i < n then { yield(i); upto(i + 1, n) }

fun to-list( action ) : <div> list<int>
  var xs := []
  with ctl yield(x){
    xs := Cons(x, xs); resume(()) // record x, then resume
  }
  action()
  xs.reverse

fun run() : <div> list<int>
  to-list({ upto(3,7) }) // [3,4,5,6]
```

The core idea (Plotkin & Pretnar, 2009): split terms into **values** v and **computations** c , and add just **two** new forms:

Values $v ::= x \mid () \mid \mathbf{true} \mid \mathbf{false} \mid \lambda x. c \mid \dots$

Comp. $c ::= \mathbf{return} \ v \mid \mathbf{op}(v; y.c) \mid v_1 \ v_2 \mid \mathbf{with} \ h \ \mathbf{handle} \ c \mid \dots$

Handlers $h ::= \{ \mathbf{return} \ x \mapsto c_r, [\mathbf{op}(x; k) \mapsto c_{op}]_{op} \}$

The core idea (Plotkin & Pretnar, 2009): split terms into **values** v and **computations** c , and add just **two** new forms:

Values $v ::= x \mid () \mid \text{true} \mid \text{false} \mid \lambda x. c \mid \dots$

Comp. $c ::= \text{return } v \mid \text{op}(v; y.c) \mid v_1 v_2 \mid \text{with } h \text{ handle } c \mid \dots$

Handlers $h ::= \{ \text{return } x \mapsto c_r, [\text{op}(x; k) \mapsto c_{op}]_{op} \}$

- **Perform** an operation, $\text{op}(v; y.c)$ – call op with argument v , bind its result to y , and **continue** as c . So $y.c$ is the **rest of the computation**.
- **Handle** it, with $h \text{ handle } c$ – the handler h gives a **return** clause (for a final value) and, per operation, a clause c_{op} using the argument x and the **resumption** k (this is resume).

Handling has just **two** rules – a **return** runs the return clause; an **operation** runs its clause, with k bound to “the rest, **re-handled**”:

$$\text{with } h \text{ handle } (\text{return } v) \quad \rightarrow \quad c_r[x \mapsto v]$$
$$\text{with } h \text{ handle } (op(v; y.c)) \quad \rightarrow \quad c_{op}[x \mapsto v, k \mapsto \lambda y. \text{with } h \text{ handle } c]$$

Handling has just **two** rules – a **return** runs the return clause; an **operation** runs its clause, with k bound to “the rest, **re-handled**”:

$$\text{with } h \text{ handle } (\text{return } v) \rightarrow c_r[x \mapsto v]$$
$$\text{with } h \text{ handle } (op(v; y.c)) \rightarrow c_{op}[x \mapsto v, k \mapsto \lambda y. \text{with } h \text{ handle } c]$$

That second rule is the whole story: op is **intercepted**, and k – the captured continuation – is handed to the clause. **How often** c_{op} **calls** k decides the effect:

0 (drop) \Rightarrow exception $1 \Rightarrow$ state, reader $\geq 2 \Rightarrow$ nondet., generator

1. Why Effect Systems?
2. Type-and-Effect Systems
3. Algebraic Effects & Handlers
Examples by Resume Count
4. Effect Systems in Practice

Tracking **what a computation does** – not just its result – now appears in many forms:

System	Mechanism	Tracks
Java	throws clauses	checked exceptions
Haskell	monads (IO , State)	effects in the type ctor
Koka	row of effect labels	exn , div , st , \dots
Eff / OCaml 5	algebraic eff. + handlers	user-defined operations
Scala ZIO	$\text{ZIO}[R, E, A]$	errors E , capabilities R
Rust	<code>async</code> , <code>unsafe</code> , <code>const</code>	coarse-grained effects

The **same** recipe everywhere: enrich the judgment to $\Gamma \vdash e : \tau ! \varepsilon$, **union** effects as they accumulate, and let **handlers discharge** them.

- Course Review

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>