Lecture 26 – Effect Systems

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- 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 $\texttt{lin}\, P \mid \texttt{un}\, P$; **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**.

Contents

Contents

A Type Says *What*, Not *What It Does*

PLRG

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/

Contents

An Effect Is a Set of Things That May Happen

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

$$\varnothing \subseteq \{\texttt{exn}\} \subseteq \{\texttt{exn}, \texttt{io}\} \subseteq \{\texttt{exn}, \texttt{io}, \texttt{st}\} \subseteq \cdots$$

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

Effects Live on the Arrow

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

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

$$\tau_1 \xrightarrow{\varepsilon} \tau_2 \qquad \text{(Koka writes this } \texttt{int -> <exn> int}\text{)}$$

Types	τ	$::=$	$\texttt{bool} \mid \texttt{int} \mid \tau_1 \xrightarrow{\varepsilon} \tau_2$
Effects	ε	$::=$	a set $\subseteq \{\texttt{exn}, \texttt{io}, \texttt{st}, \texttt{div}\}$
Terms	e	$::=$	$x \mid \lambda x{:}\tau.\, e \mid e\ e \mid \texttt{throw}\ e \mid \texttt{try}\ e\ \texttt{catch}\ e$

Typing Rules: Values Are Pure, Application Unions

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

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

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

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

Typing Rules: Raising and Handling

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

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

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

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

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

Subeffecting: Over-Approximating Is Always Safe

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

$$\frac{\Gamma \vdash e : \tau \,!\, \varepsilon \qquad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash e : \tau \,!\, \varepsilon'}\ \text{T-Sub}$$

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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \,!\, \varepsilon_1 \qquad \Gamma \vdash e_2 : \tau \,!\, \varepsilon_2 \qquad \Gamma \vdash e_3 : \tau \,!\, \varepsilon_3}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : \tau \,!\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}\ \text{T-If}$$

A Worked Example

```
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)
```

$$
\begin{array}{lcll}
\texttt{throw}(\ldots) & : & \texttt{int}\,!\,\{\texttt{exn}\} & (\text{T-Throw}) \\
\texttt{if}\ (b{=}0)\ \ldots\ (a/b) & : & \texttt{int}\,!\,\{\texttt{exn}\} & (\text{T-If}) \\
\texttt{safe-div} & : & (\texttt{int},\texttt{int}) \xrightarrow{\{\texttt{exn}\}} \texttt{int} & (\text{T-Abs}) \\
\texttt{safe-div}(10,0) & : & \texttt{int}\,!\,\{\texttt{exn}\} & (\text{T-App}) \\
\texttt{catch}(\ldots) & : & \texttt{int}\,!\,\varnothing & (\text{T-Try subtracts exn})
\end{array}
$$

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

Contents

Algebraic Effects & Handlers: the Vocabulary

PLRG

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.

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

Resume 0 Times – Exceptions

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
```

Resume 1 Time – State

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
```

Resume $\geq$ 2 Times – Nondeterminism

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 $[x]$.

Resume $\geq$ 2 Times – Generators

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]
```

Algebraic Effects, Formally

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

$$
\begin{array}{llll}
\text{Values} & v & ::= & x \mid () \mid \texttt{true} \mid \texttt{false} \mid \lambda x.\, c \mid \cdots \\
\text{Comp.} & c & ::= & \texttt{return}\ v \mid op(v;\, y.c) \mid v_1\, v_2 \mid \texttt{with}\ h\ \texttt{handle}\ c \mid \cdots \\
\text{Handlers} & h & ::= & \{\, \texttt{return}\ x \mapsto c_r,\ \ [\, op(x;\, k\,) \mapsto c_{op}\,]_{op}\,\}
\end{array}
$$

- **Perform** an operation, $op(v;\, y.c)$ – 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, $\texttt{with}\ h\ \texttt{handle}\ c$ – 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: the Resumption k *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**":

$$\texttt{with}\ h\ \texttt{handle}\ (\texttt{return}\ v) \quad \to \quad c_r[\,x \mapsto v\,]$$

$$\texttt{with}\ h\ \texttt{handle}\ (op(v;\, y.c)) \quad \to \quad c_{op}\big[\,x \mapsto v,\quad k \mapsto \lambda y.\,\texttt{with}\ h\ \texttt{handle}\ c\,\big]$$

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 $\qquad$ $1 \Rightarrow$ state, reader $\qquad$ $\geq 2 \Rightarrow$ nondet., generator

Contents

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`, ...
Eff / OCaml 5	algebraic eff. + handlers	user-defined operations
Scala ZIO	`ZIO[R,E,A]`	errors E, capabilities R
Rust	`async`, `unsafe`, `const`	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.

Next Lecture

- Course Review

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