A Type System for Effect Handlers and Dynamic Labels

Paulo Emílio de Vilhena and François Pottier

25/04/2023

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Overview

Type Systems. In this paper, we propose *Tes*, a *type system* for *effect handlers*.

Semantics of Handlers. We also explore the different choices in the *design space of handlers*. We argue in favor of one particular *interface* for programming with handlers.

Semantics of Handlers

Effect Handlers - 101

Effect handlers generalize *exception handlers*:

Whereas *raising* an exception *discards* the computation,
performing an effect *suspends* the computation, which is reified as a *continuation*.

```
exception Division_by_zero

let ( / ) x y =
 if y = 0 then raise Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | exception Division_by_zero -> 0
  | y -> y
```

```
effect Division_by_zero : int

let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
     continue k 0
  | y -> y
```

(*Examples in OCaml 4.12*)

Effect Handlers - 101

Effect handlers generalize *exception handlers*:

Whereas *raising* an exception *discards* the computation,
performing an effect *suspends* the computation, which is reified as a *continuation*.

```
exception Division_by_zero

let ( / ) x y =
 if y = 0 then raise Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | exception Division_by_zero -> 0
  | y -> y
```

```
-: int = 0
```

```
effect Division_by_zero : int

let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
     continue k 0
  | y -> y
```

(*Examples in OCaml 4.12*)

Effect Handlers - 101

Effect handlers generalize *exception handlers*:

Whereas *raising* an exception *discards* the computation,
performing an effect *suspends* the computation, which is reified as a *continuation*.

```
exception Division_by_zero

let ( / ) x y =
 if y = 0 then raise Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | exception Division_by_zero -> 0
  | y -> y
```

```
-: int = 0
```

```
effect Division_by_zero : int

let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
     continue k 0
  | y -> y
```

```
-: int = 1
```

(*Examples in OCaml 4.12*)

Effect Names

An *effect name* specifies which effect is handled by a handler.

In the previous example, the effect name is `Division_by_zero`.

It is *globally defined*: its scope spans over the entire program.

```
effect Division_by_zero : int

let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
     continue k 0
  | y -> y
```

Effect Names

An *effect name* specifies which effect is handled by a handler.

In the previous example, the effect name is `Division_by_zero`.

It is *globally defined*: its scope spans over the entire program.

```
effect Division_by_zero : int

let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y

let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
      continue k 0
  | y -> y
```

We also argue in favor of *locally defined* names.

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Allocate a memory cell named `calls`*.*

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Apply `ff` *to a modified version of* `f` *that performs* `Tick` *when called.*

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Increment `calls` *by one when* `Tick` *is performed.*

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Read the state of `calls` *at the end.*

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

This implementation however is *incorrect*!

Effect Names

Specification. *The function* `counter` *counts the number of times* `ff` *calls its argument.*

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

There are *two* problems with this implementation of `counter`:

1. The function `ff` might *intercept* `Tick` effects.
2. The function `f` might *perform* `Tick` effects.

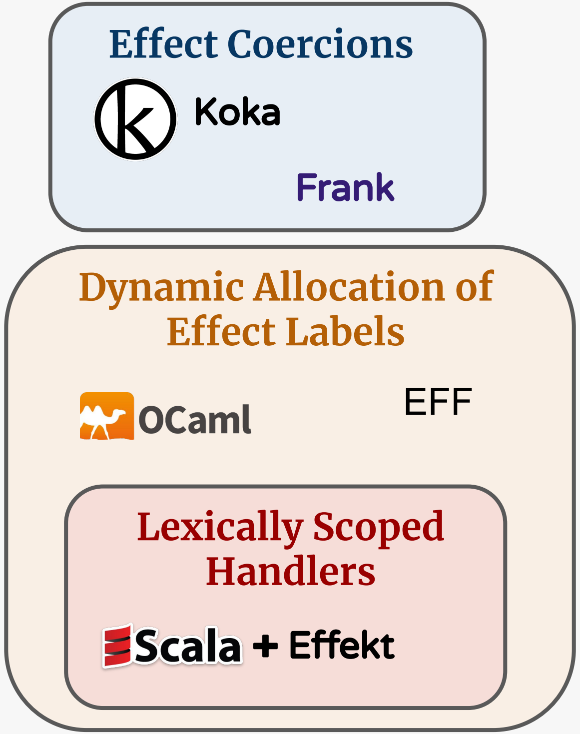
Panorama of Semantics of Handlers

There are at least *three* approaches to overcome the issue that `f` might perform `Tick` effects:

1. ***Effect Coercions***
   Allow an effect to *bypass* its innermost handler.

2. ***Dynamic Allocation of Effect Labels***
   Allows an effect to be *locally defined*.

3. ***Lexically Scoped Handlers***
   Combine *effect allocation* and *handler* into a single operation, a *lexically scoped handler*.



1. Effect Coercions

Koka's `mask` allows an effect to bypass its innermost handler.

```
effect ctl tick() : ()

fun counter(ff :  forall <e> (a -> e b) -> e  c)
               : (forall <e> (a -> e b) -> e (c, int))
  fn(f) {
    val comp =
      with ctl tick() {fn(n) {resume(())(n + 1)}}
      val y =
        ff (fn(x) {tick(); mask<tick>(fn() {f(x)})})
      fn(n) {(y, n)}
    comp(0)
  }
```

Koka

1. Effect Coercions

Koka's `mask` allows an effect to bypass its innermost handler.

```
effect ctl tick() : ()

fun counter(ff :  forall <e> (a -> e b) -> e  c)
               : (forall <e> (a -> e b) -> e (c, int))
  fn(f) {
    val comp =
      with ctl tick() {fn(n) {resume(())(n + 1)}}
      val y =
        ff (fn(x) {tick(); mask<tick>(fn() {f(x)})})
      fn(n) {(y, n)}
    comp(0)
  }
```

Koka

Convenience. Operational semantics and type systems for effect coercions have been extensively studied (Biernacki et al.).

Limitation. Coercions modify the mechanism with which an effect finds its handler.

2. Dynamic Allocation of Effect Labels

In OCaml, an *effect declaration* binds an *effect name* to a *fresh effect label*.
Its *scope* can be either *global* or *local*.

```
effect Tick : unit

let counter ff f =
  let calls = ref 0 in

  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

OCaml

2. Dynamic Allocation of Effect Labels

In OCaml, an *effect declaration* binds an *effect name* to a *fresh effect label*.
Its *scope* can be either *global* or *local*.

```
let counter ff f =
  let calls = ref 0 in
  let effect Tick : unit in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

2. Dynamic Allocation of Effect Labels

In OCaml, an *effect declaration* binds an *effect name* to a *fresh effect label*.
Its *scope* can be either *global* or *local*.

```
let counter ff f =
  let calls = ref 0 in
  let effect Tick : unit in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Convenience. It is the standard semantics of OCaml and it is similar to the approach used for *exceptions* in OCaml and ML.

Limitation. No type system (yet!). Devising such a system is the topic of this paper.

3. Lexically Scoped Handlers

The idiom of allocating an effect and immediately installing its handler is known as a *lexically scoped handler*.

The Scala library Effekt is restricted to this flavor of handler.

```
def counter[A,B,C](ff:  [E] => (A => B / E) =>  C         / E)
                      : ([E] => (A => B / E) => (C, Int) / E) =
  [E] => (f: A => B / E) =>
    var calls = 0
    handle {(scope : Scope[_, E]) =>
      val t = new Tick {
        type effect = scope.effect
        def tick() = scope.switch {resume =>
          calls = calls + 1; resume(())}
      }
      {ff(x => t.tick() andThen f(x))} map {y => (y, calls)}
    }
```

Scala + Effekt

3. Lexically Scoped Handlers

The idiom of allocating an effect and immediately installing its handler is known as a *lexically scoped handler*.

The Scala library Effekt is restricted to this flavor of handler.

```
def counter[A,B,C](ff:  [E] => (A => B / E) =>  C         / E)
                      : ([E] => (A => B / E) => (C, Int) / E) =
  [E] => (f: A => B / E) =>
    var calls = 0
    handle {(scope : Scope[_, E]) =>
      val t = new Tick {
        type effect = scope.effect
        def tick() = scope.switch {resume =>
          calls = calls + 1; resume(())}
      }
      {ff(x => t.tick() andThen f(x))} map {y => (y, calls)}
    }
```

Scala + Effekt

3. Lexically Scoped Handlers

Convenience. There are *multiple type systems* for lexically scoped handlers.

Limitation. Lexically scoped handlers impose a *“capability-passing” style*.

```
def drunkFlip(amb: Amb, exc: Exc) =
  for {
    caught ← amb.flip()
    heads ← if (caught) amb.flip() else exc.raise("We dropped the coin")
  } yield if (heads) "Heads" else "Tails"
```

Scala + Effekt

(Example from Brachthäuser et al. – JFP’20)

This Paper

We argue in favor of the *dynamic allocation of effect labels*.

And we introduce *Tes*,
a type system for *effect handlers* and *dynamic labels*.

In the next part of the talk, I am going to show

1. What is the *standard approach* in systems for effects.
2. What is the *challenge* in devising a system for dynamic labels.
3. What is the *key idea* of Tes.
4. What are the *interesting aspects* of the system, *typing* and *subtyping rules*.

Dynamic Allocation of Effect Labels

OCaml EFF

Tes

Syntax of Types

Tes follows the standard approach of type systems with support for effects:
to annotate an arrow type with a *row*.
In Tes, a row describes the effects that a function might *perform* or *handle*.

```
τ, κ ::= …
       | τ -{ρ}-> κ          (Annotated Arrow)
       | ∀α. τ               (Value Polymorphism)
       | ∀θ. τ               (Effect Polymorphism)

   ρ ::= <>                  (Empty Row)
       | (E:τ=>κ).ρ          (Effect Signature)
       | (E:Abs).ρ           (Absence Signature)
       | θ·ρ                 (Row Variable)
```

Example

The function `filter` yields the elements of `xs` that satisfy the function `p`.

```
let rec filter xs p =
  match xs with
  | [] -> ()
  | x :: xs ->
    (if p x then perform (Yield x));
    filter xs p
```

```
filter : ∀α. ∀θ.
  α list ->
  (α -{θ}-> bool) -{Ƴ[α].θ}->
  unit
  where Ƴ[α] = Yield:α=>unit
```

Reading.

"For every set of effects `θ`*, if* `p` *performs effects in* `θ`*,*
then the expression `filter xs p` *performs effects in* `Ƴ[α].θ`*."*

Example

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†   = Yield:Abs
    and γ[α] = Yield:α=>unit
```

Example

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†    = Yield:Abs
    and γ[α] = Yield:α=>unit
```

By instantiating α with `int` and θ with `<>` (the empty row), `reassemble` can be used to handle the following application of `filter`:

```
reassemble (fun () -> filter [0; 1; 2] (fun x -> x mod 2 = 0))
```

Example

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†   = Yield:Abs
    and γ[α] = Yield:α=>unit
```

By instantiating `α` with `int` and `θ` with `<>` (the empty row),
`reassemble` can be used to handle the following application of `filter`:

```
reassemble (fun () -> filter [0; 1; 2] (fun x -> x mod 2 = 0))
```

```
-: int list = [0; 2]
```

A Problem with Name Collisions?

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†    = Yield:Abs
    and γ[α] = Yield:α=>unit
```

Wait! Can `θ` be instantiated to `γ[_]`?
In other words, can the substitution of `θ` introduce a *name collision*?

A Problem with Name Collisions?

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†   = Yield:Abs
    and γ[α] = Yield:α=>unit
```

Wait! Can `θ` be instantiated to `γ[_]`?
In other words, can the substitution of `θ` introduce a *name collision*?

```
let unsafe : unit -{γ†.γ[unit]}-> int list =
  fun () -> reassemble (fun () -> perform (Yield 0); perform (Yield ()))
```

A Problem with Name Collisions?

The function `reassemble` installs a handler that *accumulates* the elements yielded by `prog`.

```
let reassemble prog =
  match prog() with
  | effect (Yield x) k ->
      x :: continue k ()
  | () -> []
```

```
reassemble : ∀α. ∀θ.
  (unit -{γ[α].θ}-> unit) -{γ†.θ}->
  α list
  where γ†    = Yield:Abs
    and γ[α] = Yield:α=>unit
```

Wait! Can `θ` be instantiated to `γ[_]`?
In other words, can the substitution of `θ` introduce a *name collision*?

```
let unsafe : unit -{γ†.γ[unit]}-> int list =
  fun () -> reassemble (fun () -> perform (Yield 0); perform (Yield ()))
```

Our answer is *Yes*. The function `unsafe`, for instance, is *well-typed!*

The Key Idea

Key idea. *To guard a function type with the assumption that names are distinct.*

More specifically, we change the usual reading of an arrow type

```
f : τ -{ρ}-> κ
```

This type now adds the *absence of name collisions* in **ρ** as a *precondition* to the evaluation of `f`.

The Key Idea

Key idea. *To guard a function type with the assumption that names are distinct.*

More specifically, we change the usual reading of an arrow type

```
f : τ -{ρ}-> κ
```

This type now adds the *absence of name collisions* in **ρ** as a *precondition* to the evaluation of `f`.

New Reading.

“If the names in **ρ** *are distinct, then, when applied to a value of type* **τ***, the function* `f`

(1) returns a value of type **κ** *(or diverges);*

(2) and, in the meantime, might perform one or more of the effects in **ρ***.”*

The Key Idea

Key idea. *To guard a function type with the assumption that names are distinct.*

```
unsafe : unit -{Y†.Y[unit]}-> int list
```

```
let unsafe() =
  reassemble (fun () ->
    perform (Yield 0); perform (Yield ())
  )
```

The Key Idea

Key idea. *To guard a function type with the assumption that names are distinct.*

```
unsafe : unit -{γ†.γ[unit]}-> int list  ≃  empty -> int list
```

```
let unsafe() =
  reassemble (fun () ->
    perform (Yield 0); perform (Yield ())
  )
```

The type `empty` has no inhabitant, thus `unsafe` cannot be called.

The Interesting Bits

Typing Judgment.

$$\Gamma \vdash e : \rho : \tau$$

Reading.

"*Under the assumption that names in* ρ *are distinct*,

the evaluation of the expression e

(1) returns a value of type τ *(or diverges);*

(2) and, in the meantime,

might perform one or more of the effects in ρ*.*"

The Interesting Bits

Typing Rules.

$$\frac{\Gamma \vdash \texttt{e} : (\texttt{E}:\textbf{Abs}).\rho : \tau}{\Gamma \vdash \textbf{let}\ \texttt{effect}\ \texttt{E}\ \textbf{in}\ \texttt{e} : \rho : \tau}\ \textit{(Effect)}$$

Reading (Bottom-Up).

"The allocation of the effect E

(1) allows e *to install a handler for this effect,*

(2) allows e *to assume that* E *is distinct from names in* **ρ**.*"*

The Interesting Bits

Subtyping Rules.

$$\frac{}{\tau \ \text{-}\{\rho\}\text{->}\ \kappa \quad \leq \quad \tau \ \text{-}\{\rho'.\rho\}\text{->}\ \kappa} \ (\textit{Extend})$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \text{ is a permutation of } \rho_2}{\tau \ \text{-}\{\rho_1\}\text{->}\ \kappa \quad \leq \quad \tau \ \text{-}\{\rho_2\}\text{->}\ \kappa} \ (\textit{Permute})$$

Is it sound to discard the permission to install a handler?

$$\frac{D \vdash E \ \# \ \rho}{D \vdash \tau \ \text{-}\{(E\text{:Abs}).\rho\}\text{->}\ \kappa \quad \leq \quad \tau \ \text{-}\{\rho\}\text{->}\ \kappa} \ (\textit{Erase})$$

The Interesting Bits

Subtyping Rules.

$$\frac{}{\tau \text{ -\{}\rho\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho'.\rho\text{\}-> } \kappa} \; (Extend)$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \text{ is a permutation of } \rho_2}{\tau \text{ -\{}\rho_1\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho_2\text{\}-> } \kappa} \; (Permute)$$

Is it sound to discard the permission to install a handler?

$$\frac{D \vdash E \;\#\; \rho}{D \vdash \tau \text{ -\{(E:Abs).}\rho\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho\text{\}-> } \kappa} \; (Erase)$$

The Interesting Bits

Subtyping Rules.

$$\frac{}{\texttt{τ -\{ρ\}-> κ} \quad \leq \quad \texttt{τ -\{ρ'.ρ\}-> κ}} \; \textit{(Extend)}$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \textit{ is a permutation of } \rho_2}{\texttt{τ -\{}\rho_1\texttt{\}-> κ} \quad \leq \quad \texttt{τ -\{}\rho_2\texttt{\}-> κ}} \; \textit{(Permute)}$$

Is it sound to discard the permission to install a handler?

$$\frac{D \vdash \texttt{E \# ρ}}{D \vdash \texttt{τ -\{(E:Abs).ρ\}-> κ} \quad \leq \quad \texttt{τ -\{ρ\}-> κ}} \; \textit{(Erase)}$$

The Interesting Bits

Subtyping Rules.

$$\frac{}{\tau\ \text{-}\{\rho\}\text{->}\ \kappa \quad \leq \quad \tau\ \text{-}\{\rho'.\rho\}\text{->}\ \kappa}\ (\textit{Extend})$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \text{ is a permutation of } \rho_2}{\tau\ \text{-}\{\rho_1\}\text{->}\ \kappa \quad \leq \quad \tau\ \text{-}\{\rho_2\}\text{->}\ \kappa}\ (\textit{Permute})$$

$$\frac{D \vdash E\ \#\ \rho}{D \vdash \tau\ \text{-}\{(E\text{:Abs}).\rho\}\text{->}\ \kappa \quad \leq \quad \tau\ \text{-}\{\rho\}\text{->}\ \kappa}\ (\textit{Erase})$$

Is it sound to discard the permission to install a handler?

The Interesting Bits

Subtyping Rules.

$$\frac{}{\texttt{τ -\{ρ\}-> κ} \quad \leq \quad \texttt{τ -\{ρ'.ρ\}-> κ}} \ (\textit{Extend})$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \text{ is a permutation of } \rho_2}{\tau \ \texttt{-\{}\rho_1\texttt{\}->} \ \kappa \quad \leq \quad \tau \ \texttt{-\{}\rho_2\texttt{\}->} \ \kappa} \ (\textit{Permute})$$

$$\frac{D \vdash E \ \# \ \rho}{D \vdash \ \texttt{τ -\{(E:Abs).ρ\}-> κ} \quad \leq \quad \texttt{τ -\{ρ\}-> κ}} \ (\textit{Erase})$$

Is it sound to discard the permission to install a handler?

No! Removing E also removes the assumption that E is distinct from names in ρ.

The Interesting Bits

Subtyping Rules.

$$\frac{}{\tau \text{ -\{}\rho\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho' \cdot \rho\text{\}-> } \kappa} \text{(Extend)}$$

A concise and powerful rule that allows a row to be (arbitrarily) extended with new entries. If a collision is introduced, the type is unusable.

Because entries are supposedly distinct, their order in a row is not important.

$$\frac{\rho_1 \text{ is a permutation of } \rho_2}{\tau \text{ -\{}\rho_1\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho_2\text{\}-> } \kappa} \text{(Permute)}$$

$$\frac{D \vdash E \ \# \ \rho}{D \vdash \tau \text{ -\{(E:Abs).}\rho\text{\}-> } \kappa \quad \leq \quad \tau \text{ -\{}\rho\text{\}-> } \kappa} \text{(Erase)}$$

D *is a disjointness context, it stores pairs of distinct names.*

Conclusion

Conclusion

Semantics of Handlers.

- Through the example of `counter`,
  we argued that the standard semantics of *global effect names* is *unsatisfactory*.

- We explored the *panorama of semantics of handlers* known in the literature:
  1. Effect coercions
  2. *Dynamic allocation of effect labels*
  3. Lexically scoped handlers

  And we argued in favor of the second option, which is currently adopted by *OCaml 5*.

Conclusion

Type Systems.

- We introduced *Tes*, a type system for *effect handlers* and *dynamic labels*.
- In doing so we had faced a *name-collision* problem: *effect names might collide*.
- Our key idea is to modify the usual reading of an arrow type

  ```
  f : τ -{ρ}-> κ
  ```

  To include the *absence of name collisions* in **ρ** as a *precondition* to the evaluation of `f`.
- We showed how *powerful typing* and *subtyping* rules can then be succinctly stated.

Metatheory.

- We have omitted the *metatheory* of Tes from this talk. Check out the paper to know:

1. What are the *guarantees* of Tes. (No unhandled effects.)
2. How we articulate its *proof of soundness*.
3. What is the relation between *effect polymorphism* and *absence of accidental handling*.

Questions

Proof of Soundness

Our proof of soundness follows the *semantic approach*, which consists of three steps:

1. Translate typing judgments as *specifications* written in a *certain program logic*.
   (In our case, we choose *TesLogic*, a *Separation Logic* with support for *handlers*.)

2. Prove that, if a *typing judgment* is *derivable*, then its *translation holds*.

3. Show that the translation implies the *system's desired guarantees*.

Pictorially,

$$\Gamma \vdash e : \rho : \tau \quad \Longrightarrow \quad \Gamma \vDash e : \rho : \tau$$

"e *is well-typed*"

"e *can be verified*; *it satisfies a specification*"

$$\vDash e : \langle\rangle : \text{unit} \quad \Longrightarrow$$

"e *is safe*: *every operation, including an effect, is well-defined*"

Effect Parametricity & Absence of Accidental Handling

The literature suggests that *parametricity of effect polymorphism* is *equivalent* to the *absence of accidental handling*.

Parametricity of Effect Polymorphism.

System S *enjoys parametric effect polymorphism* if

∃ *model* of S

1. A *logic* (*prop*, ∀, ∃, ∧, ∨, ...)
2. An *interpretation of types*
   $\mathbb{V}$: ... → *type* → (*val* → *prop*)
3. A *semantic domain of rows*
   SRow

...

such that

$$\mathbb{V}[\![\forall\theta.\ \tau]\!]_{\eta} \;=\; \forall E : \mathit{SRow}.\ \mathbb{V}[\![\tau]\!]_{\eta\{\theta \to E\}}$$

Absence of Accidental Handling.

System S *enjoys absence of accidental handling* if *Zhang and Myers*'s *equivalence laws* hold.

In particular,

```
ff : ∀θ. (int -{θ}-> int) -{θ}-> int
```

```
ff (fun x -> 2*x)    ≃

  let effect E in
  match ff (fun x -> perform (E x)) with
  | effect (E x) k -> continue k (2*x)
  | y -> y
```

Effect Parametricity & Absence of Accidental Handling

Let *Tes+TryFinally* be *Tes* extended with a *try-finally* construct, `try e finally f`, which executes the finally branch ***f*** every time control leaves ***e***.

The system *Tes+TryFinally* enjoys *parametric effect polymorphism*, yet it *does not* enjoy *absence of accidental handling*.

```
let ff : ∀θ. (int -{θ}-> int) -{θ}-> int =
  fun f ->
    let r = ref 0 in
    try let _ = f 0 in !r finally (r := !r + 1)
```

```
ff (fun x -> 2*x)    ≄    let effect E in
                          match ff (fun x -> perform (E x)) with
                          | effect (E x) k -> continue k (2*x)
                          | y -> y
```

`ff (fun x -> 2*x)` ↙ 0

`let effect E in …` ↘ 1

Handler Rule

The Interesting Bits

Typing Rules.

$$\frac{\Gamma \vdash e : (E{:}\iota{=}{>}\kappa).\rho : \tau \qquad \Gamma,\ y{:}\tau \vdash \boldsymbol{r} : (E{:}\mathbf{Abs}).\rho : \tau' \qquad \Gamma,\ x{:}\iota,\ k{:}\kappa\text{-}\{\rho\}\text{->}\tau' \vdash \boldsymbol{h} : (E{:}\mathbf{Abs}).\rho : \tau'}{\Gamma \vdash \begin{array}{l}\texttt{match e with}\\ \texttt{| effect (E x) k -> } \boldsymbol{h}\\ \texttt{| y -> } \boldsymbol{r}\end{array} : (E{:}\mathbf{Abs}).\rho : \tau'}\ (\textit{Handler})$$

Reading.

"Given the permission to install a handler `E:`**`Abs`***, the handlee* `e` *is allowed to perform* `E` *according to an arbitrary signature* `E:`**`ι=>κ`***, provided that*

(1) ***r*** *is well-typed,*

(2) ***h*** *is well-typed w.r.t this signature."*

Type-Checking `counter`

Type-Checking `counter`

$$\vdash \texttt{counter} : \langle\rangle : \forall \alpha\, \beta\, \gamma.\ (\forall \theta_1.\ (\alpha \xrightarrow{\{\theta_1\}} \beta) \xrightarrow{\{\theta_1\}} \gamma) \to (\forall \theta_2.\ (\alpha \xrightarrow{\{\theta_2\}} \beta) \xrightarrow{\{\theta_2\}} (\gamma * \mathrm{int}))$$

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking `counter`

$$\vdash \boxed{} : \langle\rangle : \quad \forall \alpha\ \beta\ \gamma.\ (\forall \theta_1.\ (\alpha \text{ -}\{\theta_1\}\text{-> } \beta) \text{ -}\{\theta_1\}\text{-> } \gamma) \text{ -> } (\forall \theta_2.\ (\alpha \text{ -}\{\theta_2\}\text{-> } \beta) \text{ -}\{\theta_2\}\text{-> } (\gamma * \text{int}))$$

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking counter

$\alpha, \beta, \gamma,$
$ff : \forall\theta_1.(\alpha\text{-}\{\theta_1\}\text{->}\beta)\text{-}\{\theta_1\}\text{->}\gamma \vdash \boxed{} : \text{<>} : \forall\theta_2.\ (\alpha\ \text{-}\{\theta_2\}\text{->}\ \beta)\ \text{-}\{\theta_2\}\text{->}\ (\gamma * \text{int})$

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking counter

α, β, γ,
ff : ∀θ₁.(α-{θ₁}->β)-{θ₁}->γ,
θ₂,
f : α-{θ₂}->β

$\vdash$ [ ] : θ_2 ; γ * int

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking counter

α, β, γ,
ff : $\forall\theta_1.(\alpha\text{-}\{\theta_1\}\text{->}\beta)\text{-}\{\theta_1\}\text{->}\gamma$,
θ_2,
f : $\alpha\text{-}\{\theta_2\}\text{->}\beta$,
calls : int ref

$\vdash$ [ ] : θ_2 ; γ * int

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking counter

$\alpha, \beta, \gamma,$
ff : $\forall\theta_1.(\alpha\text{-}\{\theta_1\}\text{->}\beta)\text{-}\{\theta_1\}\text{->}\gamma,$
$\theta_2,$
f : $\alpha\text{-}\{\theta_2\}\text{->}\beta,$
calls : int ref

⊢ ☐ : (Tick:Abs) · θ_2 ; γ * int

It is sound to assume that Tick *does not collide with* θ_2*:* f *does not perform* Tick *effects.*

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking counter

α, β, γ,
ff : $\forall\theta_1.(\alpha\text{-}\{\theta_1\}\text{->}\beta)\text{-}\{\theta_1\}\text{->}\gamma$,
θ_2,
f : $\alpha\text{-}\{\theta_2\}\text{->}\beta$,
calls : int ref

$\vdash$ [ ] : $(\text{Tick:unit=>unit})\cdot\theta_2$; γ

Specialize the type of ff *with*
$\theta_1 := (\text{Tick:unit=>unit})\cdot\theta_2$

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

Type-Checking `counter`

α, β, γ,
ff : ∀θ₁.(α-{θ₁}->β)-{θ₁}->γ,
θ₂,
f : α-{θ₂}->β,
calls : int ref

⊢ [ ] : <> ; α -{(Tick:unit=>unit)·θ₂}-> β

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```

f : α-{θ₂}->β
≤
f : α-{(Tick:unit=>unit)·θ₂}->β

Type-Checking `counter`

$$\vdash \texttt{counter} : \langle\rangle : \forall \alpha\, \beta\, \gamma.\ (\forall \theta_1.\ (\alpha \xrightarrow{\{\theta_1\}} \beta) \xrightarrow{\{\theta_1\}} \gamma) \to (\forall \theta_2.\ (\alpha \xrightarrow{\{\theta_2\}} \beta) \xrightarrow{\{\theta_2\}} (\gamma * \mathrm{int}))$$

```
let counter ff = fun f ->
  let calls = ref 0 in
  let open struct effect Tick : unit end in
  match ff (fun x -> perform Tick; f x) with
  | effect Tick k ->
      calls := !calls + 1; continue k ()
  | y -> (y, !calls)
```