Modal Effect Types

Wenhao Tang

The University of Edinburgh

EHOP Workshop, 18th Aug 2025

(Joint work with Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, Anton Lorenzen)

A Recap of Traditional Effect Types 📚

a function of type `A →{E} B` may perform effects `E` when applied

Pure Functions

A first-order pure function

```
inc : Int ->{} Int
inc x = x + 1
```

A higher-order pure function

```
app1 : (Int ->{} 1) * Int ->{} 1
app1 (f, x) = f x
```

Effects

An effect with one operation

```
effect Gen = yield : Int ⇒ 1
```

A first-order effectful function

```
gen : Int →{Gen} 1
gen x = do yield x
```

A higher-order effectful function

```
app2 : (Int →{Gen} 1) * Int →{Gen} 1
app2 (f, x) = f x
```

More Effects

Another effect

```
effect Reader = ask : 1 ⇒ Int
```

Another first-order effectful function

```
askAndGen : 1 →{Reader, Gen} 1
askAndGen () = do gen (do ask ())
```

Another higher-order effectful function

```
app3 : (Int →{Reader} 1) * Int →{Reader} 1
app3 (f, x) = f x
```

Effect Polymorphism

We have three application functions so far

```
app1 : (Int →{}       1) * Int →{}       1
app2 : (Int →{Gen}    1) * Int →{Gen}    1
app3 : (Int →{Reader} 1) * Int →{Reader} 1
```

Abstracting them by an **effect variable** `e`

```
app : ∀ e . (Int →{e} 1) * Int →{e} 1
```

Effect-polymorphic versions of `inc`, `gen`, and `askAndGen`

```
inc       : ∀ e . Int →{e}              Int
gen       : ∀ e . Int →{Gen, e}         1
askAndGen : ∀ e . 1   →{Reader, Gen, e} 1
```

Handlers

A handler for the `Gen` effect

```
asList : ∀ e . (1 →{Gen, e} 1) →{e} List Int
asList f = handle f () with
  return () ⇒ nil
  yield x r ⇒ cons x (r ())
```

Running

```
>>> asList (fun () → gen 42; gen 37)  -- recall that gen x = do yield x
[42, 37] : List Int
```

Composing Handlers

A handler for the `Reader` effect

```
answer : ∀ e . (1 →{Reader, e} 1) →{e} 1
answer f = handle f () with
  return x ⇒ x
  ask () r ⇒ r 42
```

Compose the two handlers to handle `askAndGen`

```
>>> asList (fun () → answer askAndGen) -- recall that askAndGen () = do gen (do ask ())
[42] : List Int
```

Modal Effect Types in 1️⃣ Page

decoupling effect annotations from function arrows

Traditional effect types

Effects **entangled** with function arrows

First-order functions

```
inc       : ∀ e . Int →{e}                Int
gen       : ∀ e . Int →{Gen, e}           1
askAndGen : ∀ e . 1   →{Reader, Gen, e} 1
```

Higher-order functions

```
app : ∀ e . (Int →{e} 1) * Int →{e} 1
map : ∀ e . (A →{e} B) * List A →{e} List B
```

Handlers

```
asList : ∀ e . (1 →{Gen, e}     1) →{e} List Int
```

Modal effect types

Decouple effects from arrows via modalities

Use [E] to specify the effects being used

```
inc       :               [](Int → Int)
gen       :            [Gen](Int → 1)
askAndGen : [Reader, Gen](1   → 1)
```

As app itself is pure, we use []

```
app : []((Int → 1) * Int → 1)
map : []((A → B) * List A → List B)
```

Use <E> to specify the effects being handled

```
asList : [](   <Gen>(1 → 1) → List Int)
```

Inspired by **Frank** which is based on an elaboration from (almost) the right to the left

> Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. POPL 2017.

and **Effekt** which distinguishes between first-class and second-class functions and adopts a capability-passing translation

> Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. OOPSLA 2020.

Modal effect types (**MET**) have a completely new foundation, multimodal type theory (Gratzer et al. 2020), and smoothly support first-class functions

```
-- Traditional effect types
inc       : ∀ e . Int →{e}                Int
gen       : ∀ e . Int →{Gen, e}           1
askAndGen : ∀ e . 1   →{Reader, Gen, e} 1
app       : ∀ e . (Int →{e} 1) * Int →{e} 1
```

```
-- Modal effect types
inc       :              [](Int → Int)
gen       :           [Gen](Int → 1)
askAndGen : [Reader, Gen](1   → 1)
app       : []((Int → 1) * Int → 1)
```

Diving into Modal Effect Types 🤿

modes, modalities, locks 🔒, and keys 🔑

Effect Contexts (Modes)

Effect contexts track effects provided by the context

```
⊢    fun x . do yield x    :    Int → 1    @    Gen
--   ~~~~~~~~~~~~~~~~~~         ~~~~~~~~         ~~~~~
--   term                       type             effect context
--   @ Gen                      @ Gen
```

Variables in the context also share the ambient effect context

```
     f : Int → 1  ⊢  fun x . f x  :  Int → 1   @   Gen
--   ~~~~~~~~~~~~~     ~~~~~~~~~~~~     ~~~~~~~~~
--   @ Gen             @ Gen            @ Gen
```

Subeffecting happens naturally

```
⊢    fun x . do yield x    :    Int → 1    @    Gen, Reader
```

Modalities

An **absolute modality** `[E]` changes the ambient effect context to `E` (`[E](F) = E`)

```
--  modality introduction introduces a lock         [Gen] changes F to Gen
--  ~~~~~~~~~~                                      ~~~~~~
    🔒_[Gen]   ⊢   fun x . do yield x  :  Int → 1  @ Gen
──────────────────────────────────────────────────────────────────
    ⊢ mod_[Gen] (fun x . do yield x)  :  [Gen](Int → 1)  @ F
--     ~~~~~~~~~~
--     modality introduction
```

A **relative modality** `<E>` adds effects `E` to the ambient effect context (`<E>(F) = E,F`)

```
--  modality introduction introduces a lock                   <Gen> changes Reader to Gen, Reader
--  ~~~~~~~~~~                                                ~~~~~~~~~~~~~~
    🔒_<Gen>   ⊢   fun x . do yield (do ask ())  :  Int → 1  @ Gen, Reader
──────────────────────────────────────────────────────────────────
    ⊢ mod_<Gen> (fun x . do yield (do ask ()))  :  <Gen>(Int → 1)  @ Reader
```

Locks 🔒

Locks control the accessibility of variables

An invalid judgement

```
❌  f : Int → 1  ⊢  mod_[Gen] (fun x . f x)  :  [Gen](Int → 1)  @  Reader
--  ~~~~~~~~~~~~              ~~~~~~~~~~~~~
--  @ Reader                  @ Gen
```

Its premise does not hold

```
❌   f : Int → 1, 🔒_[Gen]  ⊢  fun x . f x  :  Int → 1  @  Gen
--  ~~~~~~~~~~~~  ~~~~~~~~
--  @ Reader      disallows f to be used
--                changes Reader to Gen (reading from right to left)
```

Modality Elimination

We can make the premise well-typed by annotating the binding of `f` with `[]`

```
    f :_[] Int → 1, 🔒_[Gen]  ⊢  fun x . f x  :  Int → 1  @  Gen
--           ~~~~~~~~~
--           @ . (because [] changes Reader to .)
--     ~~~~~~~~~~~~~~~~
--     @ Reader
```

Such a binding is introduced by **modality elimination** (the default annotation is `<>`)

```
    f :_[] Int → 1  ⊢  M  :  A  @  E
  ─────────────────────────────────────────────────
    ⊢  let mod_[] f = mod_[] (fun x → ()) in M  :  A  @  E
--       ~~~~~~~~~~~
--       let-style modality elimination
```

Modality elimination can be inferred in practice

Modality Transformations 🔑

How does typing decide that `f :_[] Int → 1` can be used after `🔒 _[Gen]` ?

```
--                                        ⭐ a modality transformation, the key to 🔒 _[Gen]
--                                                                    ~~~~~~~~~~~~~~~~~~~~~
f :_[] Int → 1, 🔒 _[Gen]  ⊢  fun x . f x  :  Int → 1  @  Gen         [] ⇒ [Gen] @ Reader
-----------------------------------------------------------------------------------------
f :_[] Int → 1  ⊢  mod_[Gen] (fun x . f x)  :  [Gen](Int → 1)  @  Reader
```

A **modality transformation** `μ ⇒ ν @ E` is the key to unlock `🔒 _ν` for variable bindings of form `f :_μ A`

Soundness: `μ ⇒ ν @ E` must guarantee that `μ(F) ≤ ν(F)` for all `E ≤ F`

`[] ⇒ [Gen] @ Reader` obviously satisfies the soundness condition

Handlers

Back to the `asList` handler

```
asList : [](<Gen>(1 → 1) → List Int)
asList f = handle f () with return () ⇒ nil | yield x r ⇒ cons x (r ())
```

It is elaborated to

```
asList : [](<Gen>(1 → 1) → List Int)
asList = mod_[] (fun f → let mod_<Gen> f' = f in handle f' () with return () ⇒ nil | yield x r ⇒ cons x
--        ~~~~~~            ~~~~~~~~~~~~~~~~~~~~~
--        introduction      elimination
```

A handler also introduces a lock with a relative modality

```
--                         introduced by the handler which handles Gen
--                         ~~~~~~~~~
    f' :_<Gen> 1 → 1, 🔒_<Gen>  ⊢  f' ()  :  1  @ Gen, E        <Gen> ⇒ <Gen> @ E        ...
```

Modality Transformation Rules

Suppose our effect contexts are **scoped rows**, i.e.,

- Duplicated effects are allowed
- Reordering is allowed except for identical effects

We have the following rules for modality transformations

1. `[E1] ⇒ [E2] @ E3` if `E1 ⩽ E2`
2. `[E1] ⇒ <E2> @ E3` if `E1 ⩽ E2, E3`
3. `<E1> ⇒ <E2> @ E3` if `E1 ═ E2`
4. `<E1> ⇒ [E2] @ E3` impossible

Rule 1️⃣

`[E1] ⇒ [E2] @ E3` if `E1 ⩽ E2`

If some term only uses effects `E1`, we can use it at a larger effect context

```
   f :_[] 1 → 1, 🔒_[Gen]  ⊢  f () : 1 @ Gen  -- [] ⇒ [Gen] @ E
--         ~~~~~~
--         @ .                                  we have . ⩽ Gen regardless of E
-- ~~~~~~~~~~~~~~
-- @ E
```

Otherwise we may leak effects

Rule 2️⃣

`[E1] ⇒ <E2> @ E3` if `E1 ⩽ E2, E3`

If some term only uses effects `E1`, we can use it at a larger effect context

```
   f :_[Reader] 1 → 1, 🔒_<Gen>  ⊢  f () : 1 @ Gen, Reader, E  -- [Reader] ⇒ <Gen> @ Reader
--                  ~~~~~~~
--                  @ Reader                                  we have Reader ⩽ Gen, Reader,
-- ~~~~~~~~~~~~~~~~~~~
-- @ Reader, E
```

Otherwise we may leak effects

Rule 4️⃣

`<E1> ⇒ [E2] @ E3` impossible

If a term uses `E1` plus the ambient effects `E3`, we can never use it at some fixed `E2`

Because `E3` can be arbitrarily upcasted

```
   f :_<Gen> 1 → 1,    🔒_[Gen] ⊢ f () : 1 @ Gen  -- ❌ <Gen> ⇒ [Gen] @ E
--          ~~~~~~~~
--          @ Gen, E                                 we do not have Gen,E ≤ E for any E
-- ~~~~~~~~~~~~~~~~~~
-- @ E
```

In terms of traditional effect types, we want to upcast `1 →{Gen, e} 1` (where `e` is from the typing context) to `∀ e' . 1 →{Gen, e'} 1`

Rule 3️⃣

<E1> ⇒ <E2> @ E3 if E1 == E2

```
  f :_<Gen> 1 → 1, 🔒_<Gen> ⊢ f () : 1 @ Gen, E    -- <Gen> ⇒ <Gen> @ E
```

But what's wrong with <> ⇒ <Gen>? **Accidental handling**

Otherwise, we could give the following type to `asList`

```
asList : []((1 → 1) → List Int)
asList f = handle f () with return () ⇒ nil | yield x r ⇒ cons x (r ())
```

This type does not reflect the fact that `Gen` used by the argument is handled

With parameterised effects, e.g., having both `Gen Int, Gen Bool`, this could lead to a crash

Masking

`[]((1 → 1) → List Int)` is a bad type for `asList`

But sometimes we do want to conceal the internal implementation

```
--        <Gen> is required for well-typedness but leaks implementation details
--        ~~~~~~~~~~~~~
  find' : <Gen>(Int → Bool) → List Int → Maybe Int
  find' p xs = handle (iterate (fun x → if p x then do yield x else ()) xs) with
    return _   ⇒ nothing,
    yield  x _ ⇒ just x
```

Solution: mask the `Gen` effect for `p x`

```
  find : (Int → Bool) → List Int → Maybe Int
  find p xs = handle (iterate (fun x → if mask<Gen>(p x) then do yield x else ()) xs) with
--                                        ~~~~~~~~~~~~~~
--                                        mask removes Gen from the effect context
```

The Masking Modality

In addition to `<E>`, we also have the **masking modality** `<E▷`.

In fact, relative modalities have the general form `<E1|E2>`

Relative modalities act on effect contexts as `<E1|E2>(E) = E2 + (E - E1)`

A `mask<E>` introduces a lock `🔒_<E▷`

```
--                        does not change E
--  @ E                   <Gen>∘<Gen▷ = <>         @ E
--  ~~~~~~~~~~~~~~        ~~~~~~~~~~~~~~~~~~~      ~~~~
    p : Int → Bool, x : Int, 🔒_<Gen>, 🔒_<Gen▷  ⊢  p x   :  Bool  @  E
─────────────────────────────────────────────────────────────────────────
    p : Int → Bool, x : Int, 🔒_<Gen>  ⊢           mask<Gen>(p x)  :  Bool  @  Gen, E
--  ~~~~~~~~~~~~~~        ~~~~~~~~~                ~~~~~~~~~~~~~~
--  @ E                   changes E to Gen, E      @ Gen, E
```

More Examples

to show the ergonomics of modal effect types

Cooperative Concurrency 🧵

Smoothly store effectful functions into datatypes

```
effect Coop = ufork : 1 ⇒ Bool | suspend : 1 ⇒ 1

data Proc = proc (List Proc → 1)

push     : [](Proc → List Proc → List Proc) -- push a process into a queue
next     : [](List Proc → 1)                -- run the first process in the queue
schedule : [](<Coop>(1 → 1) → List Proc → 1)
schedule m = handle m () with
  return  ()   ⇒ fun q → next q,
  suspend () r ⇒ fun q → next (push (proc (r ())) q),
  ufork   () r ⇒ fun q → r true (push (proc (r false)) q)
```

In contrast, with traditional effect types we have to deal with effect variables

```
data Proc e = proc (List Proc →{e} 1)
push : ∀ e . Proc →{e} List Proc →{e} List Proc
```

Re-generating

Pre-process all generated numbers with a function.

```
--         used by regen          handled by regen
--         ~~~~~                  ~~~~~
  regen : [Gen]((Int → Int) → <Gen>(1 → 1) → 1)
  regen f m = handle m () with
    return () ⇒ ()
    yield s r ⇒ do yield (f s); r ()
--                 ~~~~~~~~~~~~~~
--                 the handler itself uses yield, thus we have [Gen] instead of [] for the whole
```

In contrast, with traditional effect types we have

```
--                                                      one handled, one used
--                                                      ~~~~~~~~~~~~~
  regen : ∀ e . (Int →{Gen, e} Int) →{} (1 →{Gen, Gen, e} 1) →{Gen, e} 1
```

More Features

parametric polymorphism, kinds

Parametric Polymorphism

Completely unsurprising.

Polymorphic `app`

```
app : ∀ a b . []((a → b) * a → b)
```

More higher-order functions

```
map     : ∀ a b . []((a → b)  → List a → List b)
iterate : ∀ a   . []((a → ()) → List a → ())
```

and handlers

```
answer : ∀ [a] . [](<Reader>(1 → a) → a)
```

Hang on, what is `∀ [a]` ???

Kinds

`∀ [a]` is a shorthand for `∀ a:Abs`

We distinguish types independent of the ambient effect context from others

- **Absolute types** (kind `Abs`)
  built from base types, positive types, and types boxed by an absolute modality
  (e.g., `Int`, `Bool`, `[](Int → Int)`, `Int * (Bool + [](Int → Int))`)
  cannot leak effects

- **Unrestricted types** (kind `Any`)
  also include functions not boxed by an absolute modality
  (e.g., `Int → Int`, `Bool * (1 → 1)`, `<Gen>(Int → Bool)`)
  can leak effects

Subkinding `Abs ⩽ Any`

Returning from Handlers

Why do we need to restrict `a` to kind `Abs` for `answer`?

```
answer : ∀ [a] . [](<Reader>(1 → a) → a)
```

Consider returning a function of type `1 → Int` from a handler

```
  handle (fun () → do ask ()) with return x ⇒ x | ask () r ⇒ r 42
--        ~~~~~~~~~~~~~~~~~~~~~      ~~~~~~~~~~~~~~
--        returns a function         captured by x
```

We cannot give it type `1 → Int` – otherwise the usage of `ask` is untracked!

One solution is to restrict the return value to have kind `Abs`

Or we can give a more general type to `answer`

Effect Polymorphism

Consider the polymorphic map

```
map : ∀ a b e . (a →{e} b) →{} List a →{e} List b
```

Almost all of the time the following is good enough

```
map : []((a → b) → List a → List b)
```

but still misses the information that the middle arrow is pure

We can recover it by going back to effect variables

```
map : ∀ e . []([e](a → b) → [e](List a → List b))
```

Effect variables are particularly helpful for higher-order effects

Higher-Order Effects

Operation argument and result types must have kind `Abs`

If we allowed `effect Leak = leak : (1 → 1) ⇒ 1`, then we could write

```
--                                              expected to be handled by asList
--                                              ~~~~~~~~~~~~
   handle asList (fun () → do leak (fun () → do yield 42)) with
     return _ ⇒ fun () → 37
     leak p _ ⇒ p
--   ~~~~~~~~~~~~~~
--   p is substituted with (fun () → do yield 42) ➡️ yield escapes the scope of asList
```

which leaks the yield operation

We need to use `effect Leak = leak : [E](1 → 1) ⇒ 1` with a specific `E`

or be parameterised over `E`: `effect Leak e = leak : [e](1 → 1) ⇒ 1`

Encoding Effect Types

a unified framework for studying and comparing effect types

Encoding Rows à la Koka

$$
\begin{aligned}
[\![A \to\{E\}\, B]\!] &= [\![E]\!]([\![A]\!] \to [\![B]\!]) \\
[\![\forall a.A]\!] &= \forall a.[\![A]\!]
\end{aligned}
$$

Encoding Capabilities à la Effekt

$$
\begin{aligned}
[\![(\overline{A}, \overline{f : T}) \Rightarrow B]\!] &\quad = \quad \forall \overline{f^*}.\langle \overline{f^*} \rangle (\overline{[\![A]\!]} \to \overline{[f^*][\![T]\!]} \to [\![B]\!]) \\
[\![T \ \mathbf{at}\ C]\!] &\quad = \quad [[\![C]\!]][\![T]\!]
\end{aligned}
$$

Summary 🗒️

More in the Papers

Modal Effect Types. OOPSLA 2025.

Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen.

- MET: a core calculus with modal effect types
- Type soundness, effect safety
- A simple bidirectional typing algorithm which infers all modality introduction and elimination
- An encoding of a fragment of traditional effect types into MET without effect variables

Rows and Capabilities as Modal Effects. Draft. Wenhao Tang and Sam Lindley.

- MET(X): parameterised by an **effect theory** X following Rose (Morris and McKinna 2019)
- An extension with local labels for encoding named handlers
- Encoding various Koka's core calculi into MET(R_{scp})
- Encoding various Effekt's core calculi into MET(S)

Ongoing and Future Work

- On types:
  - novel bidirectional typing for both first-class polymorphism and modal types
  - higher-order effects without effect variables by tracking the order of handlers
  - Fitch-style modality elimination
- On semantics:
  - denotational semantics / logical relations
- Generally:
  - absolute and relative modalities for variable contexts instead of effect contexts (could be useful to multi-stage programming)
  - a general account for sub-moding in MTT

Takeaways

- **Decouple effects from function arrows**
- Do not annotate every function arrow with its effects – annotate effects once for the whole function and then annotate only when there is a change of effects
- By doing so, effect variables become unnecessary (mostly)

```
-- Traditional effect types
inc       : ∀ e . Int →{e}                Int
gen       : ∀ e . Int →{Gen, e}           1
askAndGen : ∀ e . 1   →{Reader, Gen, e} 1
app       : ∀ e . (Int →{e} 1) * Int →{e} 1
asList    : ∀ e . (1 →{Gen, e}     1) →{e} List Int
answer    : ∀ e . (1 →{Reader, e} 1) →{e} 1
map      : ∀ a b e. (a →{e} b) →{} List a →{e} List b
iterate  : ∀ a e. (a →{e} 1) →{} List a →{e} 1
schedule : ∀e.(1 →{Coop,e} 1) →{e} List (Proc e) →{e} 1
regen    : ∀ e . (Int →{Gen, e} Int)
             →{} (1 →{Gen, Gen, e} 1) →{Gen, e} 1
```

```
-- Modal effect types
inc       :             [](Int → Int)
gen       :          [Gen](Int → 1)
askAndGen : [Reader, Gen](1   → 1)
app       : []((Int → 1) * Int → 1)
asList    : [](    <Gen>(1 → 1) → List Int)
answer    : [](<Reader>(1 → 1) → 1)
map      : ∀ a b. []((a → b) → List a → List b)
iterate  : ∀ a. []((a → 1) → List a → 1)
schedule : [](<Coop>(1 → 1) → List Proc → 1)
regen    : [Gen]((Int → Int) → <Gen>(1 → 1) → 1)
```