Modal Effect Types

Wenhao Tang[1], Leo White[2], Stephen Dolan[2], Daniel Hillerström[1], Sam Lindley[1], Anton Lorenzen[1]

OOPSLA, 19th Oct 2025

1 THE UNIVERSITY of EDINBURGH

2 Jane Street

A Recap of Traditional Effect Types 📋

Effects and Effect Types

Effects are the way programs interact with their environment

Including I/O, concurrency, exceptions, nondeterminism, probability

Effect types statically track use of effects in types

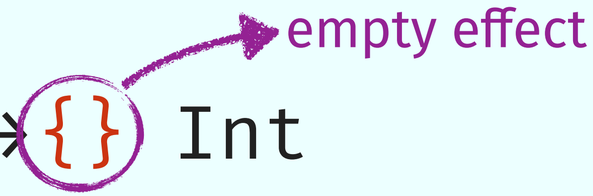
Traditional Effect Types

```
inc : Int → Int
inc x = x + 1


app : (Int → Int) → Int → Int
app f x = f x
```

▶️ `app inc 42`

```
43 : Int
```

Traditional Effect Types




```
inc : Int →{} Int
inc x = x + 1
```

traditional effect types annotate each function arrow with the effects it uses

```
app : (Int →{} Int) →{} Int →{} Int
app f x = f x
```

Traditional Effect Types

```
inc : Int →{} Int
inc x = x + do ask ()
```

`ask : 1 ⇒ Int` takes a unit and returns an integer

```
app : (Int →{} Int) →{} Int →{} Int
app f x = f x
```

Traditional Effect Types

```
inc : Int →{ask} Int
inc x = x + do ask ()

app : (Int →{} Int) →{} Int →{} Int
app f x = f x
```

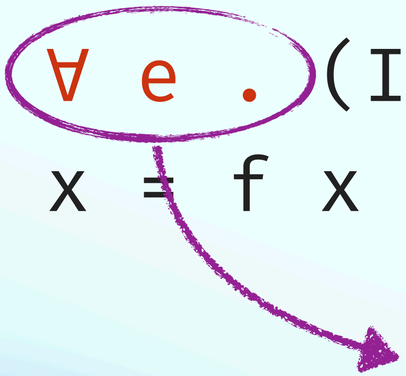
Traditional Effect Types

```
inc : Int →{ask} Int
inc x = x + do ask ()

app : (Int →{ask} Int) →{ask} Int →{ask} Int
app f x = f x
```

we also need to annotate function arrows of app
to allow effectful arguments

Traditional Effect Types

```
inc : Int →{ask} Int
inc x = x + do ask ()

app : ∀ e . (Int →{e} Int) →{e} Int →{e} Int
app f x = f x
```



the most general type requires **effect polymorphism**

Traditional Effect Types

```
inc : ∀ e . Int →{ask, e} Int
inc x = x + do ask ()

app : ∀ e . (Int →{e} Int) →{e} Int →{e} Int
app f x = f x

ans : ∀ e . (1 →{ask, e} Int) →{e} Int
ans f = handle f () with { ask () r ⟼ r 37 }
```

handler's argument is allowed to use ask
plus any other effects abstracted by e

a handler of ask which resumes the continuation r with 37

Traditional Effect Types

```
inc : ∀ e . Int →{ask, e} Int
app : ∀ e . (Int →{e} Int) →{e} Int →{e} Int
ans : ∀ e . (1 →{ask, e} Int) →{e} Int
```

Effect variables are verbose

Adding traditional effect types to existing languages requires significant rewriting of type signatures such as app

Can we get rid of these effect variables?

Our Proposal: Modal Effect Types 💡

Modal Effect Types

👀 Let us take a closer look at app

```
app : ∀ e . (Int →{e} Int) →{e} Int →{e} Int
app f x = f x
```

😠 This is unfair. *The* app *itself does not even mention any effects!*

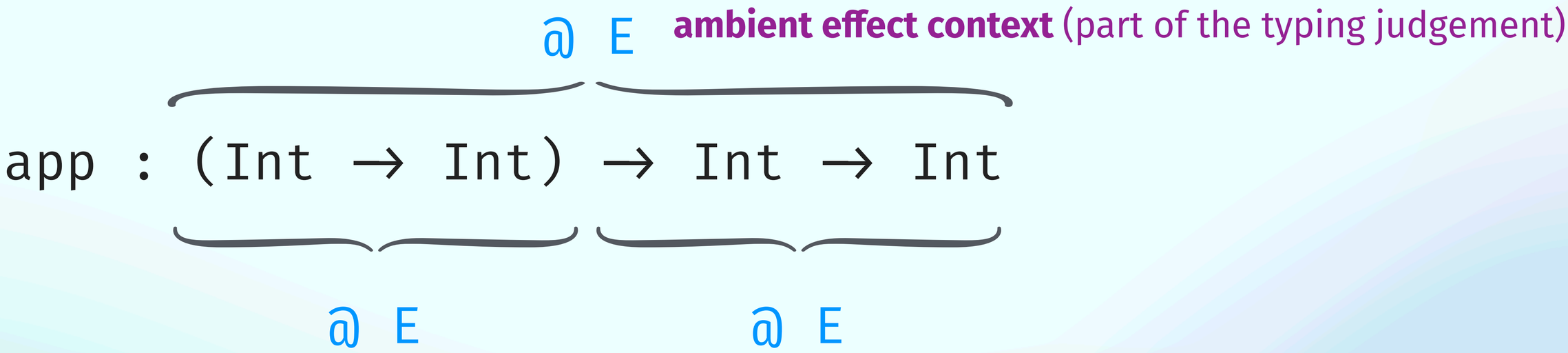
🤔 Why don't we just track this fact?

🥲 But effect types are **entangled** with function arrows in a traditional effect system

💡 Let's **decouple** effect types from function arrows

Ambient Effect Context

Typing judgements track **effect contexts**, i.e., effects provided by the context



Both the argument and result of app share the same effect context E

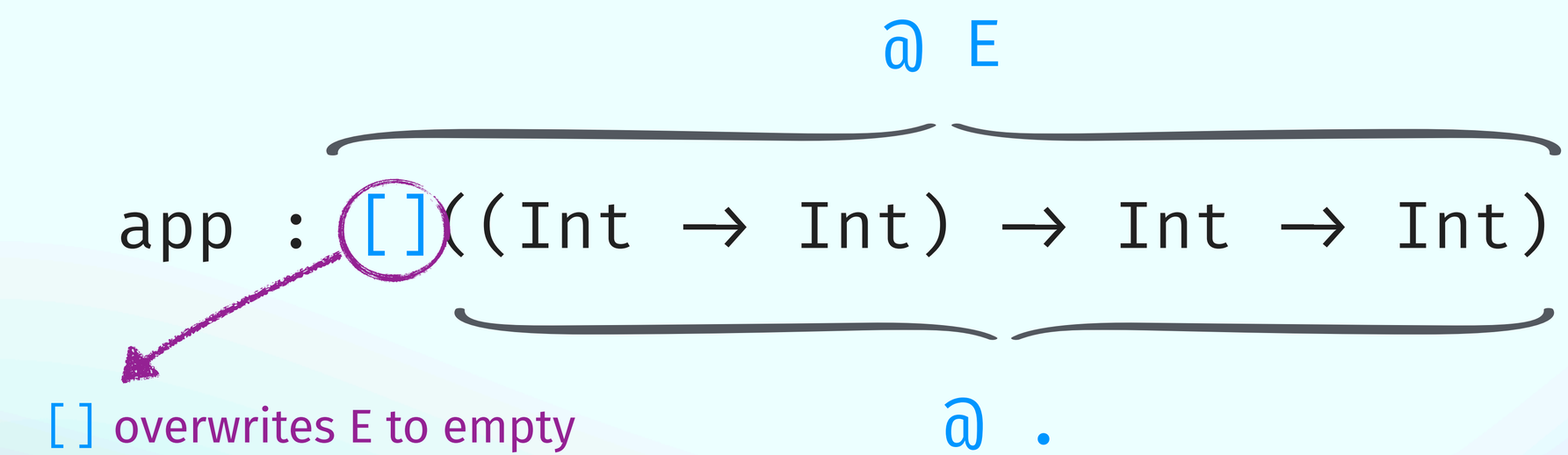
Absolute Modality

An **absolute modality** `[F]` specifies an effect context `F` in a type

```
app : (Int → Int) → Int → Int
```

Absolute Modality

An **absolute modality** `[F]` specifies an effect context `F` in a type



The empty **absolute modality** `[]` indicates that app uses no effects

By *subeffecting* we can still apply app in any effect context E

Absolute Modality

Similarly for `inc`

```
inc : ∀ e . Int →{ask, e} Int
inc x = x + do ask ()
```

Absolute Modality

Similarly for `inc`

```
inc : [ask](Int → Int)
inc x = x + do ask ()
```

The **absolute modality** `[ask]` indicates that `inc` uses `ask`

By *subeffecting* we can still apply inc in any effect context E containing `ask`

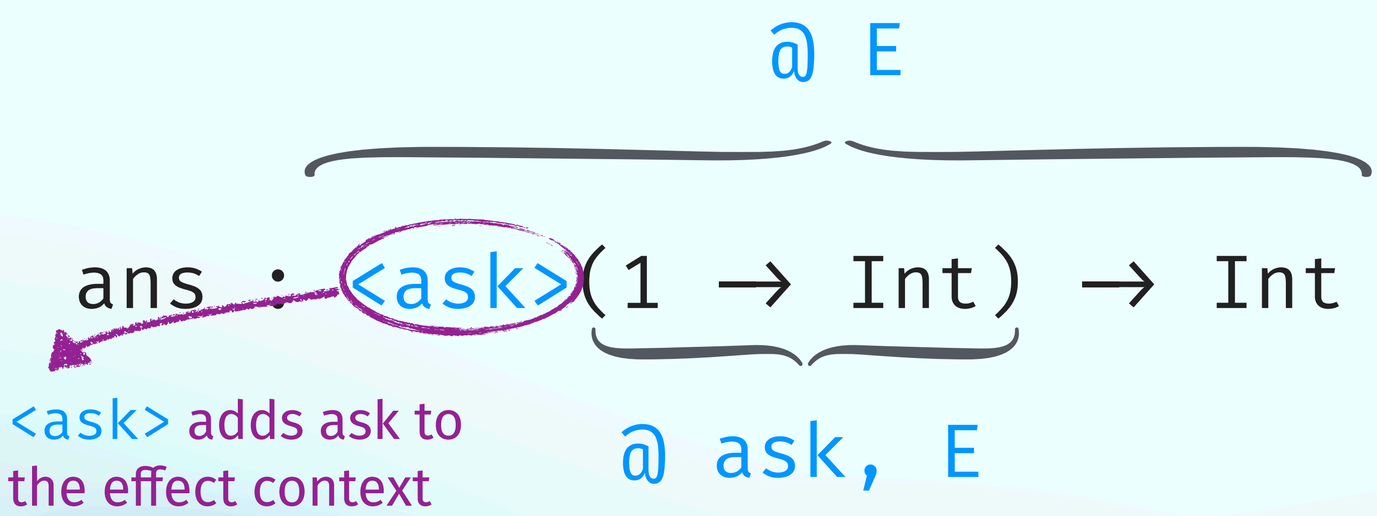
Relative Modality

A **relative modality** <F> specifies an extension F to the effect context

```
ans : ∀ e . (1 →{ask, e} Int) →{e} Int
```

Relative Modality

A **relative modality** <F> specifies an extension F to the effect context



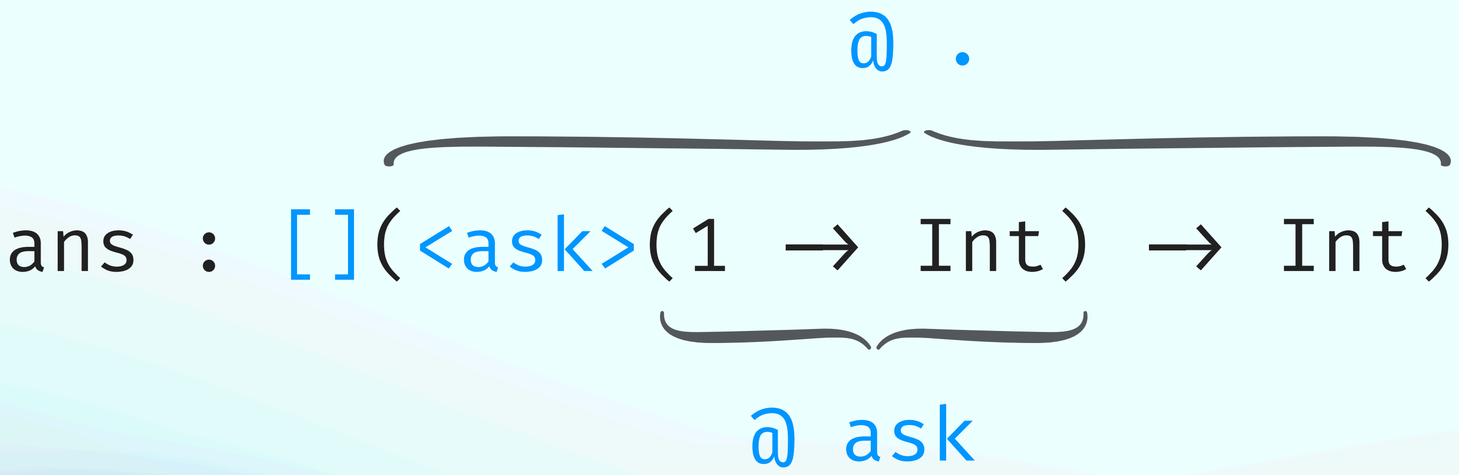
The **relative modality** <ask> indicates that ans' argument may *additionally* use ask

meanwhile can still use effects E from the context

In other words, ans **handles** ask

Relative Modality

A **relative modality** <F> specifies an extension F to the effect context



The **absolute modality** [] indicates that ans itself does not use effects

By sub-effecting we may still use ans with other effects

Modal Effect Types

```
inc : [ask](Int → Int)
app : []((Int → Int) → Int → Int)
ans : [](<ask>(1 → Int) → Int)
```

Modal Effect Types

```
inc : [ask](Int → Int)
app : (Int → Int) → Int → Int
ans : <ask>(1 → Int) → Int
```

Convention: global function definitions implicitly have a `[]`

Modal Effect Types

```
inc : [ask](Int → Int)
app : (Int → Int) → Int → Int
ans : <ask>(1 → Int) → Int
```

Compose them together

```
▶️ ans (fun () → app inc 42)
79 : Int
```

A Case Study:
Cooperative Concurrency 🧵

Cooperative Concurrency 🧵

With traditional effect types:

```
effect Coop = fork : 1 ⇒ Bool | suspend : 1 ⇒ 1
data Proc e = proc (List (Proc e) →{e} 1)

-- push a process into a queue
push : ∀ e . Proc →{e} List Proc →{e} List Proc

-- run the first process in the queue
next : ∀ e . List Proc →{e} 1

schedule : ∀ e . (1 →{Coop, e} 1) →{e} List Proc →{e} 1
schedule m = handle m () with
  return  ()    ⇒ fun q → next q,
  suspend () r ⇒ fun q → next (push (proc (r ())) q),
  fork    () r ⇒ fun q → r true (push (proc (r false)) q)
```

Cooperative Concurrency 🧵

With modal effect types:

```
effect Coop = fork : 1 ⇒ Bool | suspend : 1 ⇒ 1
data Proc = proc (List Proc → 1)

-- push a process into a queue
push : [](Proc → List Proc → List Proc)

-- run the first process in the queue
next : [](List Proc → 1)

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),
  fork    () r ⇒ fun q → r true (push (proc (r false)) q)
```

A Quick Look at the Core Calculus 👀

A Tale of Locks 🔒 and Keys 🔑

mod introduces a modality and a lock

$$\frac{\Gamma, \text{🔒}\_[\mathsf{ask}] \vdash \mathbf{fun}\ x \to f\ x : \mathsf{Int} \to \mathsf{Int}\ @\ \mathsf{ask}}{\Gamma \vdash \mathbf{mod}\_[\mathsf{ask}]\ (\mathbf{fun}\ x \to f\ x) : [\mathsf{ask}](\mathsf{Int} \to \mathsf{Int})\ @\ E}$$

the ambient effect context is overwritten to ask

locks control usage of variables

modality transformation: the key 🔑 to the lock

$$\frac{[] \Rightarrow [\mathsf{ask}]}{f :\_[]\ \mathsf{Int} \to \mathsf{Int}, \text{🔒}\_[\mathsf{ask}] \vdash f : \mathsf{Int} \to \mathsf{Int}\ @\ \mathsf{ask}}$$

$$\frac{\Gamma \vdash V : [](\mathsf{Int} \to \mathsf{Int})\ @\ E \qquad \Gamma, f :\_[]\ \mathsf{Int} \to \mathsf{Int} \vdash M : A\ @\ E}{\Gamma \vdash \mathbf{let\ mod}\_[]\ f = V\ \mathbf{in}\ M : A\ @\ E}$$

let mod eliminates a modality and introduces a binder with a modality

More in the Paper

More in the Paper

- MET: a core calculus with modal effect types
  - Based on **multimodal type theory** (Gratzer et al. 2020)
  - Inspired by languages **Frank** (Lindley et al. 2017) and **Effekt** (Brachthäuser et al. 2020)
- Masking <E ▷ (the full syntax of relative modality is <E | F>)
- Type soundness and effect safety
- Extensions to MET: parametric polymorphism, ADT
- Simple bidirectional typing for MET
- Encoding a *fragment* of traditional effect types into MET without the requirement of effect variables

Ongoing and Future Work

- Rows and Capabilities as Modal Effects.
  - a uniform framework for encoding different effect systems
  - conditionally accepted by POPL
- Better type inference for modal types
- Formal comparison with capture tracking of Scala
- Denotational semantics and logical relations
- Higher-order effects

Takeaway

Do not track effects on every function arrow -- track them when needed!

```
inc : ∀ e . Int →{ask, e} Int

app : ∀ e . (Int →{e} Int) →{e} Int →{e} Int

ans : ∀ e . (1 →{ask, e} Int) →{e} Int

data Proc e = proc (List (Proc e) →{e} 1)

push : ∀ e . Proc →{e} List Proc →{e} List Proc

next : ∀ e . List Proc →{e} 1

schedule : ∀ e . (1 →{Coop, e} 1) →{e} List Proc →{e} 1
```

Takeaway

Do not track effects on every function arrow -- track them when needed!

```
inc : [ask](Int → Int)
app : []((Int → Int) → Int → Int)
ans : [](<ask>(1 → Int) → Int)
data Proc = proc (List Proc → 1)
push : [](Proc → List Proc → List Proc)
next : [](List Proc → 1)
schedule : [](<Coop>(1 → 1) → List Proc → 1)
```

(P.S. I'm looking for a postdoc position)