Modal effect types

Sam Lindley

The University of Edinburgh

FATA Seminar, Glasgow, 18th March 2025

Joint work with
Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Anton Lorentzen

Effect polymorphism

A prototypical pure higher-order function:

`map : ∀ a b.(a → b) → List a → List b`

Effect polymorphism

A prototypical pure higher-order function:

$\texttt{map : } \forall \texttt{ a b.(a} \rightarrow \texttt{b)} \rightarrow \texttt{List a} \rightarrow \texttt{List b}$

We can only pass pure functions to `map`.

An effect-polymorphic version:

$\texttt{map' : } \forall \texttt{ a b e.(a} \xrightarrow{\texttt{e}} \texttt{b)} \xrightarrow{\texttt{e}} \texttt{List a} \xrightarrow{\texttt{e}} \texttt{List b}$

Effect polymorphism

A prototypical pure higher-order function:

$\texttt{map : } \forall \texttt{ a b.(a} \to \texttt{b)} \to \texttt{List a} \to \texttt{List b}$

We can only pass pure functions to `map`.

An effect-polymorphic version:

$\texttt{map' : } \forall \texttt{ a b e.(a} \xrightarrow{\texttt{e}} \texttt{b)} \xrightarrow{\texttt{e}} \texttt{List a} \xrightarrow{\texttt{e}} \texttt{List b}$

Is this really necessary?

Effect polymorphism

A prototypical pure higher-order function:

$$\texttt{map} : \forall\ \texttt{a}\ \texttt{b}.(\texttt{a} \rightarrow \texttt{b}) \rightarrow \texttt{List}\ \texttt{a} \rightarrow \texttt{List}\ \texttt{b}$$

We can only pass pure functions to `map`.

An effect-polymorphic version:

$$\texttt{map'} : \forall\ \texttt{a}\ \texttt{b}\ \texttt{e}.(\texttt{a} \xrightarrow{\texttt{e}} \texttt{b}) \xrightarrow{\texttt{e}} \texttt{List}\ \texttt{a} \xrightarrow{\texttt{e}} \texttt{List}\ \texttt{b}$$

Is this really necessary?

No! In Frank the signature of `map` is syntactic sugar for the signature of `map'`.

Effect polymorphism

A prototypical pure higher-order function:

$\texttt{map : } \forall\ \texttt{a b.(a} \rightarrow \texttt{b)} \rightarrow \texttt{List a} \rightarrow \texttt{List b}$

We can only pass pure functions to `map`.

An effect-polymorphic version:

$\texttt{map' : } \forall\ \texttt{a b e.(a} \xrightarrow{\texttt{e}} \texttt{b)} \xrightarrow{\texttt{e}} \texttt{List a} \xrightarrow{\texttt{e}} \texttt{List b}$

Is this really necessary?

No! In Frank the signature of `map` is syntactic sugar for the signature of `map'`.

Key observation: almost always we need only one effect variable in a type signature

Effect polymorphism

But Frank's syntactic sugar is fragile.

Effect polymorphism

But Frank's syntactic sugar is fragile.

For instance, consider a generator iterating over a list:

```
eff Gen a = yield:a ↠ 1
```

$$\texttt{gen} : \texttt{List Int} \xrightarrow{\texttt{Gen Int}} \texttt{1}$$

```
gen xs = map (fun x → do yield x) xs; ()
```

(adjusting concrete Frank syntax for consistency with the rest of the talk)

Effect polymorphism

But Frank's syntactic sugar is fragile.

For instance, consider a generator iterating over a list:

```
eff Gen a = yield:a ↠ 1
```

$$\texttt{gen : List Int} \xrightarrow{\texttt{Gen Int}} \texttt{1}$$

```
gen xs = map (fun x → do yield x) xs; ()
```

(adjusting concrete Frank syntax for consistency with the rest of the talk)

If we forget the annotation on the arrow then Frank gives the following error message.

```
cannot unify effects e and Gen Int, £
```

Effect polymorphism

But Frank's syntactic sugar is fragile.

For instance, consider a generator iterating over a list:

```
eff Gen a = yield:a ↠ 1
```

$$\texttt{gen : List Int} \xrightarrow{\texttt{Gen Int}} \texttt{1}$$

```
gen xs = map (fun x → do yield x) xs; ()
```

(adjusting concrete Frank syntax for consistency with the rest of the talk)

If we forget the annotation on the arrow then Frank gives the following error message.

```
cannot unify effects e and Gen Int, £
```

Can we do better?

From function arrows to effect contexts

Conventional effect typing — function arrows are annotated with effects

$\vdash \ \texttt{fun (f, x)} \rightarrow \texttt{f x : ((Int} \xrightarrow{\texttt{E}} \texttt{1)} \times \texttt{Int)} \xrightarrow{\texttt{E}} \texttt{1}$

From function arrows to effect contexts

Conventional effect typing — function arrows are annotated with effects

$$\vdash \quad \texttt{fun (f, x)} \rightarrow \texttt{f x} : ((\texttt{Int} \xrightarrow{\texttt{E}} \texttt{1}) \times \texttt{Int}) \xrightarrow{\texttt{E}} \texttt{1}$$

Modal effect typing — **ambient effect context** determines effects

$$\vdash \quad \texttt{fun (}\underbrace{\texttt{f}}_{\texttt{@ E}}\texttt{, x)} \underbrace{\rightarrow \texttt{f x}}_{\texttt{@ E}} : ((\underbrace{\texttt{Int} \rightarrow \texttt{1}}_{\texttt{@ E}}) \times \texttt{Int}) \underbrace{\rightarrow \texttt{1}}_{\texttt{@ E}} \quad \texttt{@} \quad \texttt{E}$$

Effects contexts

An **effect context** `E` is a row of typed operations

Effects contexts

An **effect context** `E` is a row of typed operations

Example: `get:1 ↠ Int, put:Int ↠ 1`

Effects contexts

An **effect context** `E` is a row of typed operations

Example: `get:1 ↠ Int, put:Int ↠ 1`

Effect context rows are **scoped** (as in Frank and Koka)

- repeats are allowed (same name but possibly different signatures)
- order of repeated operations matters
- relative order of distinct operations does not matter

Modal effect types

A **mode** is an effect context

A **modality** is a transformation from one mode to another

Modal effect types

A **mode** is an effect context

A **modality** is a transformation from one mode to another

MET — simply-typed core calculus of modal effect types

Modal effect types

A **mode** is an effect context

A **modality** is a transformation from one mode to another

MET — simply-typed core calculus of modal effect types

METL — surface language for MET with: bidirectional typing for inferring introduction and elimination of modalities + algebraic data types + polymorphism

Modal effect types

A **mode** is an effect context

A **modality** is a transformation from one mode to another

MET — simply-typed core calculus of modal effect types

METL — surface language for MET with: bidirectional typing for inferring introduction and elimination of modalities + algebraic data types + polymorphism

Almost all examples in this talk use the **simply-typed** fragment of METL

Overriding the ambient context with absolute modalities

$$\vdash \ \texttt{fun x} \to \underbrace{\texttt{do yield (x + 42)}}_{@\ \texttt{yield:Int} \twoheadrightarrow \texttt{1}} : (\underbrace{\texttt{Int} \to \texttt{1}}_{@\ \texttt{yield:Int} \twoheadrightarrow \texttt{1}}) \ @\ \texttt{yield:Int} \twoheadrightarrow \texttt{1}$$

Overriding the ambient context with absolute modalities

$$\vdash \ \texttt{fun}\ \texttt{x} \to \underbrace{\texttt{do}\ \texttt{yield}\ (\texttt{x} + \texttt{42})}_{@\ \texttt{yield}:\texttt{Int}\ \twoheadrightarrow\ \texttt{1}} : (\underbrace{\texttt{Int} \to \texttt{1}}_{@\ \texttt{yield}:\texttt{Int}\ \twoheadrightarrow\ \texttt{1}})\ @\ \texttt{yield}:\texttt{Int} \twoheadrightarrow \texttt{1}$$

$$\vdash \ \texttt{fun}\ \texttt{x} \to \underbrace{\texttt{do}\ \texttt{yield}\ (\texttt{x} + \texttt{42})}_{@\ \texttt{yield}:\texttt{Int}\ \twoheadrightarrow\ \texttt{1}} : [\texttt{yield}:\texttt{Int} \twoheadrightarrow \texttt{1}](\underbrace{\texttt{Int} \to \texttt{1}}_{@\ \texttt{yield}:\texttt{Int}\ \twoheadrightarrow\ \texttt{1}})\ @\ .$$

Overriding the ambient context with absolute modalities

$$\vdash\ \texttt{fun x} \rightarrow \underbrace{\texttt{do yield (x + 42)}}_{@\ \texttt{yield:Int}\ \twoheadrightarrow\ 1}\ :\ (\underbrace{\texttt{Int} \rightarrow 1}_{@\ \texttt{yield:Int}\ \twoheadrightarrow\ 1})\ @\ \texttt{yield:Int} \twoheadrightarrow 1$$

$$\vdash\ \texttt{fun x} \rightarrow \underbrace{\texttt{do yield (x + 42)}}_{@\ \texttt{yield:Int}\ \twoheadrightarrow\ 1}\ :\ [\texttt{yield:Int} \twoheadrightarrow 1](\underbrace{\texttt{Int} \rightarrow 1}_{@\ \texttt{yield:Int}\ \twoheadrightarrow\ 1})\ @\ .$$

The **absolute modality** $[\texttt{yield:Int} \twoheadrightarrow 1]$ **overrides** the empty ambient effect context $(.)$ in the function body enabling the `yield` operation to be performed.

Overriding the ambient context with absolute modalities

$$\vdash \texttt{fun x} \rightarrow \underbrace{\texttt{do yield (x + 42)}}_{\texttt{@ yield:Int} \twoheadrightarrow \texttt{1}} : (\underbrace{\texttt{Int} \rightarrow \texttt{1}}_{\texttt{@ yield:Int} \twoheadrightarrow \texttt{1}}) \texttt{ @ yield:Int} \twoheadrightarrow \texttt{1}$$

$$\vdash \texttt{fun x} \rightarrow \underbrace{\texttt{do yield (x + 42)}}_{\texttt{@ yield:Int} \twoheadrightarrow \texttt{1}} : \texttt{[yield:Int} \twoheadrightarrow \texttt{1]}(\underbrace{\texttt{Int} \rightarrow \texttt{1}}_{\texttt{@ yield:Int} \twoheadrightarrow \texttt{1}}) \texttt{ @ .}$$

The **absolute modality** `[yield:Int ↠ 1]` **overrides** the empty ambient effect context (.) in the function body enabling the `yield` operation to be performed.

In general `[E]` overrides the ambient effect context with `E`.

Overriding the ambient context with absolute modalities

$$\vdash \ \texttt{fun x} \to \underbrace{\texttt{do yield (x + 42)}}_{@\ \texttt{yield:Int} \twoheadrightarrow 1} : (\underbrace{\texttt{Int} \to 1}_{@\ \texttt{yield:Int} \twoheadrightarrow 1}) \ @\ \texttt{yield:Int} \twoheadrightarrow 1$$

$$\vdash \ \texttt{fun x} \to \underbrace{\texttt{do yield (x + 42)}}_{@\ \texttt{yield:Int} \twoheadrightarrow 1} : [\texttt{yield:Int} \twoheadrightarrow 1](\underbrace{\texttt{Int} \to 1}_{@\ \texttt{yield:Int} \twoheadrightarrow 1}) \ @\ .$$

The **absolute modality** `[yield:Int ↠ 1]` **overrides** the empty ambient effect context (.) in the function body enabling the `yield` operation to be performed.

In general `[E]` overrides the ambient effect context with `E`.

Effect contexts given by absolute modalities percolate through the structure of a type:

- a function of type `[E](A → B)` may perform effects `E` when invoked
- elements of a list of type `[E](List (A → B))` may perform effects `E` when invoked
- a value of type `[E]Int` cannot perform any effects

Effect context abbreviations

Example:

```
eff Gen a = yield:a ↠ 1
```

- `[Gen Int]` denotes the modality `[yield:Int ↠ 1]`
- `[Gen Int, E]` denotes the modality `[yield:Int ↠ 1, E]`

Absolute modalities and higher-order functions

Iteration specialised to integer lists:

```
iter : []((Int → 1) → List Int → 1)
iter f nil         = ()
iter f (cons x xs) = f x; iter f xs
```

Absolute modalities and higher-order functions

Iteration specialised to integer lists:

```
iter : []((Int → 1) → List Int → 1)
iter f nil          = ()
iter f (cons x xs) = f x; iter f xs
```

Applying a pure higher-order function in an impure effect context:

```
⊢  iter (fun x → do yield (x + 42)) : 1 @ Gen Int
```

Absolute modalities and higher-order functions

Iteration specialised to integer lists:

```
iter : []((Int → 1) → List Int → 1)
iter f nil          = ()
iter f (cons x xs) = f x; iter f xs
```

Applying a pure higher-order function in an impure effect context:

```
⊢  iter (fun x → do yield (x + 42)) : 1 @ Gen Int
```

What happened? Bidirectional typing eliminates the `[]` modality of `iter` and then upcasts its empty effect context to the singleton effect context `Gen Int`.

Absolute modalities and higher-order functions

Iteration specialised to integer lists:

```
iter : []((Int → 1) → List Int → 1)
iter f nil          = ()
iter f (cons x xs) = f x; iter f xs
```

Applying a pure higher-order function in an impure effect context:

```
⊢  iter (fun x → do yield (x + 42)) : 1 @ Gen Int
```

What happened? Bidirectional typing eliminates the `[]` modality of `iter` and then upcasts its empty effect context to the singleton effect context `Gen Int`.

Terminology:

- **boxing** = modality introduction
- **unboxing** = modality elimination

Absolute modalities and higher-order functions

Iteration specialised to integer lists:

```
iter : []((Int → 1) → List Int → 1)
iter f nil          = ()
iter f (cons x xs) = f x; iter f xs
```

Applying a pure higher-order function in an impure effect context:

```
⊢  iter (fun x → do yield (x + 42)) : 1 @ Gen Int
```

What happened? Bidirectional typing eliminates the `[]` modality of `iter` and then upcasts its empty effect context to the singleton effect context `Gen Int`.

Terminology:

- **boxing** = modality introduction
- **unboxing** = modality elimination

In a conventional effect type system `iter` would be effect-polymorphic

$$\texttt{iter} : \forall\, \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{List Int} \xrightarrow{\texttt{e}} \texttt{1}$$

Transforming the ambient context with relative modalities

Handling the `Gen Int` effect to produce a list of integers:

```
asList f =
  handle f () with
    return ()  ⇒  nil
    yield x r  ⇒  cons x (r ())
```

What type should `asList` have?

Transforming the ambient context with relative modalities

Handling the `Gen Int` effect to produce a list of integers:

```
asList f =
  handle f () with
    return ()  ⇒  nil
    yield x r  ⇒  cons x (r ())
```

What type should `asList` have?

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

The **relative modality** `<Gen Int>` **extends** the ambient effect context.

$$\vdash \texttt{fun}\ \underbrace{\texttt{f}}_{\texttt{@ Gen Int, E}}\ \to \texttt{handle}\ \underbrace{\texttt{f ()}}_{\texttt{@ Gen Int, E}}\ \texttt{with ... : <Gen Int>(}\ \underbrace{\texttt{1} \to \texttt{1}}_{\texttt{@ Gen Int, E}}\ \texttt{)} \to \texttt{List Int @ E}$$

The effect context of `f` is `Gen Int, E`.

Transforming the ambient context with relative modalities

Handling the `Gen Int` effect to produce a list of integers:

```
asList f =
  handle f () with
    return ()  ⇒  nil
    yield x r  ⇒  cons x (r ())
```

What type should `asList` have?

$$\texttt{[](<Gen Int>(1} \to \texttt{1)} \to \texttt{List Int)}$$

The **relative modality** `<Gen Int>` **extends** the ambient effect context.

$$\vdash \texttt{fun}\ \underbrace{\texttt{f}}_{\texttt{@ Gen Int, E}} \to \texttt{handle}\ \underbrace{\texttt{f ()}}_{\texttt{@ Gen Int, E}}\ \texttt{with ... : <Gen Int>(}\ \underbrace{\texttt{1} \to \texttt{1}}_{\texttt{@ Gen Int, E}}\ \texttt{)} \to \texttt{List Int @ E}$$

The effect context of `f` is `Gen Int, E`.

In a conventional effect type system `asList` would be effect-polymorphic

$$\texttt{asList} : \forall\ \texttt{e.(1}\ \xrightarrow{\texttt{Gen Int, e}}\ \texttt{1)}\ \xrightarrow{\texttt{e}}\ \texttt{List Int}$$

Coercions between modalities

Automatic unboxing in METL allows values to be coerced between different modalities

We can extend an absolute modality:

```
⊢ fun f → f : [Gen Int](1 → 1) → [Gen Int, Gen String](1 → 1) @ E
```

Coercions between modalities

Automatic unboxing in $\textsc{Metl}$ allows values to be coerced between different modalities

We can extend an absolute modality:

$\vdash \texttt{fun f} \rightarrow \texttt{f : [Gen Int](1} \rightarrow \texttt{1)} \rightarrow \texttt{[Gen Int, Gen String](1} \rightarrow \texttt{1) @ E}$

In a conventional effect type system this corresponds to:

$\vdash \texttt{fun f} \rightarrow \texttt{f : (}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, Gen String, e}} \texttt{1)}$

Coercions between modalities

Automatic unboxing in Metl allows values to be coerced between different modalities

We can extend an absolute modality:

$$\vdash \texttt{fun f} \to \texttt{f : [Gen Int](1} \to \texttt{1)} \to \texttt{[Gen Int, Gen String](1} \to \texttt{1) @ E}$$

In a conventional effect type system this corresponds to:

$$\vdash \texttt{fun f} \to \texttt{f : (}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, Gen String, e}} \texttt{1)}$$

We cannot extend a relative modality in the same way:

$$\nvdash \texttt{fun f} \to \texttt{f : <>(1} \to \texttt{1)} \to \texttt{<Gen Int>(1} \to \texttt{1) @ E}\quad \texttt{\# Ill-typed}$$

Coercions between modalities

Automatic unboxing in METL allows values to be coerced between different modalities

We can extend an absolute modality:

$$\vdash \texttt{fun f} \to \texttt{f : [Gen Int](1} \to \texttt{1)} \to \texttt{[Gen Int, Gen String](1} \to \texttt{1) @ E}$$

In a conventional effect type system this corresponds to:

$$\vdash \texttt{fun f} \to \texttt{f : (}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, Gen String, e}} \texttt{1)}$$

We cannot extend a relative modality in the same way:

$$\nvdash \texttt{fun f} \to \texttt{f : <>(1} \to \texttt{1)} \to \texttt{<Gen Int>(1} \to \texttt{1) @ E} \quad \texttt{\# Ill-typed}$$

This would insert a fresh $\texttt{yield:Int} \twoheadrightarrow \texttt{1}$ operation which may shadow other `yield` operations in `E`.

Coercions between modalities

Automatic unboxing in METL allows values to be coerced between different modalities

We can extend an absolute modality:

$$\vdash \texttt{fun f} \to \texttt{f} : [\texttt{Gen Int}](\texttt{1} \to \texttt{1}) \to [\texttt{Gen Int}, \texttt{Gen String}](\texttt{1} \to \texttt{1})\ \texttt{@}\ \texttt{E}$$

In a conventional effect type system this corresponds to:

$$\vdash \texttt{fun f} \to \texttt{f} : (\forall\ \texttt{e}.\texttt{1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1}) \xrightarrow{\texttt{E}} (\forall\ \texttt{e}.\texttt{1} \xrightarrow{\texttt{Gen Int, Gen String, e}} \texttt{1})$$

We cannot extend a relative modality in the same way:

$$\nvdash \texttt{fun f} \to \texttt{f} : \langle\rangle(\texttt{1} \to \texttt{1}) \to \langle\texttt{Gen Int}\rangle(\texttt{1} \to \texttt{1})\ \texttt{@}\ \texttt{E}\quad \texttt{\# Ill-typed}$$

This would insert a fresh $\texttt{yield:Int} \twoheadrightarrow \texttt{1}$ operation which may shadow other `yield` operations in `E`.

In a conventional effect type system this corresponds to:

$$\nvdash \texttt{fun f} \to \texttt{f} : (\texttt{1} \xrightarrow{\texttt{E}} \texttt{1}) \xrightarrow{\texttt{E}} (\texttt{1} \xrightarrow{\texttt{Gen Int, E}} \texttt{1})\ \texttt{\# Ill-typed}$$

Coercions between modalities

An absolute modality can be coerced into the corresponding relative modality.

```
⊢ fun f → f : [Gen Int](1 → 1) → <Gen Int>(1 → 1) @ E
```

Coercions between modalities

An absolute modality can be coerced into the corresponding relative modality.

$\vdash \texttt{fun f} \rightarrow \texttt{f : [Gen Int](1} \rightarrow \texttt{1)} \rightarrow \texttt{<Gen Int>(1} \rightarrow \texttt{1) @ E}$

In a conventional effect type system this corresponds to:

$\vdash \texttt{fun f} \rightarrow \texttt{f : (}\forall\texttt{ e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(1} \xrightarrow{\texttt{Gen Int, E}} \texttt{1)}$

Coercions between modalities

An absolute modality can be coerced into the corresponding relative modality.

$$\vdash \texttt{fun f} \to \texttt{f : [Gen Int](1} \to \texttt{1)} \to \texttt{<Gen Int>(1} \to \texttt{1) @ E}$$

In a conventional effect type system this corresponds to:

$$\vdash \texttt{fun f} \to \texttt{f : (}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(1} \xrightarrow{\texttt{Gen Int, E}} \texttt{1)}$$

But the converse is not permitted

$$\nvdash \texttt{fun f} \to \texttt{f : <Gen Int>(1} \to \texttt{1)} \to \texttt{[Gen Int](1} \to \texttt{1) @ E} \quad \texttt{\# Ill-typed}$$

as the argument may also use effects from the ambient effect context `E`.

Coercions between modalities

An absolute modality can be coerced into the corresponding relative modality.

$$\vdash \texttt{fun f} \to \texttt{f : [Gen Int](1} \to \texttt{1)} \to \texttt{<Gen Int>(1} \to \texttt{1) @ E}$$

In a conventional effect type system this corresponds to:

$$\vdash \texttt{fun f} \to \texttt{f : (}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(1} \xrightarrow{\texttt{Gen Int, E}} \texttt{1)}$$

But the converse is not permitted

$$\nvdash \texttt{fun f} \to \texttt{f : <Gen Int>(1} \to \texttt{1)} \to \texttt{[Gen Int](1} \to \texttt{1) @ E} \quad \texttt{\# Ill-typed}$$

as the argument may also use effects from the ambient effect context `E`.

In a conventional effect type system this corresponds to:

$$\nvdash \texttt{fun f} \to \texttt{f : (1} \xrightarrow{\texttt{Gen Int, E}} \texttt{1)} \xrightarrow{\texttt{E}} \texttt{(}\forall\ \texttt{e.1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1)}$$

Composing handlers

State effect

```
eff State s = get:1 ↠ s, put:s ↠ 1
```

Composing handlers

State effect

```
eff State s = get:1 ↠ s, put:s ↠ 1
```

A state handler (specialised to integer state)

```
state : [](<State Int>(1 → 1) → Int → 1)
state m = handle m () with
  return x  ⇒  fun s → x
  get () r  ⇒  fun s → r s s
  put s’ r  ⇒  fun s → r () s’
```

Composing handlers

Using integer state to write a generator that yields the prefix sum of a list

```
prefixSum : [Gen Int, State Int](List Int → 1)
prefixSum xs = iter (fun x → do put (do get () + x); do yield (do get ())) xs
```

Composing handlers

Using integer state to write a generator that yields the prefix sum of a list

```
prefixSum : [Gen Int, State Int](List Int → 1)
prefixSum xs = iter (fun x → do put (do get () + x); do yield (do get ())) xs
```

We can now handle the operations of `prefixSum` by composing two handlers

```
> asList (fun () → state (fun () → prefixSum [3,1,4,1,5,9]) 0)
# [3,4,8,9,14,23] : List Int
```

Composing handlers

Using integer state to write a generator that yields the prefix sum of a list

```
prefixSum : [Gen Int, State Int](List Int → 1)
prefixSum xs = iter (fun x → do put (do get () + x); do yield (do get ())) xs
```

We can now handle the operations of `prefixSum` by composing two handlers

```
> asList (fun () → state (fun () → prefixSum [3,1,4,1,5,9]) 0)
# [3,4,8,9,14,23] : List Int
```

In a conventional effect system composing handlers requires effect polymorphism

$$\texttt{asList} : \forall\, \texttt{e}.(\texttt{1} \xrightarrow{\texttt{Gen Int, e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{List Int}$$

$$\texttt{state} : \forall\, \texttt{e}.(\texttt{1} \xrightarrow{\texttt{State Int, e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{1}$$

Storing effectful functions

First-order cooperative concurrency effect

```
eff Coop = suspend:1 ↠ 1, ufork:1 ↠ Bool
```

Storing effectful functions

First-order cooperative concurrency effect

```
eff Coop = suspend:1 ↠ 1, ufork:1 ↠ Bool
```

Recursive data type of cooperative processes

```
data Proc = proc (List Proc → 1)

push : [](Proc → List Proc → List Proc)
push x xs = xs ++ cons x nil
```

```
next : [](List Proc → 1)
next q = case q of
  nil                 → ()
  cons (proc p) ps → p ps
```

Storing effectful functions

First-order cooperative concurrency effect

```
eff Coop = suspend:1 ↠ 1, ufork:1 ↠ Bool
```

Recursive data type of cooperative processes

```
data Proc = proc (List Proc → 1)

push : [](Proc → List Proc → List Proc)
push x xs = xs ++ cons x nil
```

```
next : [](List Proc → 1)
next q = case q of
  nil              → ()
  cons (proc p) ps → p ps
```

Scheduler parameterised by a list of suspended processes

```
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)
```

Storing effectful functions

First-order cooperative concurrency effect

```
eff Coop = suspend:1 ↠ 1, ufork:1 ↠ Bool
```

Recursive data type of cooperative processes

```
data Proc = proc (List Proc → 1)

push : [](Proc → List Proc → List Proc)
push x xs = xs ++ cons x nil
```

```
next : [](List Proc → 1)
next q = case q of
  nil                →  ()
  cons (proc p) ps   →  p ps
```

Scheduler parameterised by a list of suspended processes

```
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 a conventional effect system storing effectful functions requires effect polymorphism

$$\texttt{data Proc e = proc (List Proc} \xrightarrow{\texttt{e}} \texttt{1)}$$

$$\texttt{schedule : } \forall\ \texttt{e.(1} \xrightarrow{\texttt{Coop, e}} \texttt{1)} \xrightarrow{\texttt{e}} \texttt{List (Proc e)} \xrightarrow{\texttt{e}} \texttt{1}$$

Kinds

State handler for `1 → 1` computations

```
state' : [](<State Int>(1 → (1 → 1)) → Int → (1 → 1))
```

Kinds

State handler for `1 → 1` computations

```
state' : [](<State Int>(1 → (1 → 1)) → Int → (1 → 1))
```

Unsound as this type allows effects to leak

```
state' (fun () → fun () → do put (do get () + 42)) 0 : 1 → 1
```

return clause of `state'` lets `fun () → do put (do get () + 42)` escape scope of handler

Kinds

State handler for `1 → 1` computations

```
state' : [](<State Int>(1 → (1 → 1)) → Int → (1 → 1))
```

Unsound as this type allows effects to leak

```
state' (fun () → fun () → do put (do get () + 42)) 0 : 1 → 1
```

return clause of `state'` lets `fun () → do put (do get () + 42)` escape scope of handler

Sound type signature

```
state' : [](<State Int>(1 → (1 → 1)) → Int → <State Int>(1 → 1))
```

Kinds

State handler for `1 → 1` computations

```
state' : [](<State Int>(1 → (1 → 1)) → Int → (1 → 1))
```

Unsound as this type allows effects to leak

```
state' (fun () → fun () → do put (do get () + 42)) 0 : 1 → 1
```

return clause of `state'` lets `fun () → do put (do get () + 42)` escape scope of handler

Sound type signature

```
state' : [](<State Int>(1 → (1 → 1)) → Int → <State Int>(1 → 1))
```

`state` versus `state'`:

```
state  : [](<State Int>(1 → 1)       → Int → 1)
state' : [](<State Int>(1 → (1 → 1)) → Int → <State Int>(1 → 1))
```

Kinds

State handler for `1 → 1` computations

```
state' : [](<State Int>(1 → (1 → 1)) → Int → (1 → 1))
```

Unsound as this type allows effects to leak

```
state' (fun () → fun () → do put (do get () + 42)) 0 : 1 → 1
```

return clause of `state'` lets `fun () → do put (do get () + 42)` escape scope of handler

Sound type signature

```
state' : [](<State Int>(1 → (1 → 1)) → Int → <State Int>(1 → 1))
```

`state` versus `state'`:

```
state  : [](<State Int>(1 → 1)        → Int → 1)
state' : [](<State Int>(1 → (1 → 1)) → Int → <State Int>(1 → 1))
```

- `state` cannot leak the state effect
- `state'` can leak the state effect

Kinds

- **Absolute types** (e.g. `1`, `List Int`, and `[Gen Int](List Int → 1)`)
  built from base types, positive types, and types boxed by an absolute modality —
  cannot leak effects
- **Unrestricted types** (e.g. `1 → 1`, `List Int → 1`, and `<Coop>(1 → 1)`)
  also include functions not boxed by an absolute modality —
  can leak effects

Kinds

- **Absolute types** (e.g. `1`, `List Int`, and `[Gen Int](List Int → 1)`)
  built from base types, positive types, and types boxed by an absolute modality —
  cannot leak effects
- **Unrestricted types** (e.g. `1 → 1`, `List Int → 1`, and `<Coop>(1 → 1)`)
  also include functions not boxed by an absolute modality —
  can leak effects

Kinds

- `Abs` classifies absolute types
- `Any` classifies unrestricted types

Kinds

- **Absolute types** (e.g. `1`, `List Int`, and `[Gen Int](List Int → 1)`)
  built from base types, positive types, and types boxed by an absolute modality —
  cannot leak effects
- **Unrestricted types** (e.g. `1 → 1`, `List Int → 1`, and `<Coop>(1 → 1)`)
  also include functions not boxed by an absolute modality —
  can leak effects

Kinds

- `Abs` classifies absolute types
- `Any` classifies unrestricted types

Subkinding allows absolute types to be treated as unrestricted: `Abs` $\leq$ `Any`

Type polymorphism

Polymorphic version of `iter`

```
iter : ∀(a:Any).[]((a → 1) → List a → 1)
iter {a:Any} f nil         = ()
iter {a:Any} f (cons x xs) = f x; iter {a} f xs
```

Explicit type abstractions and type applications in braces.

Type polymorphism

Polymorphic version of `iter`

```
iter : ∀(a:Any).[]((a → 1) → List a → 1)
iter {a:Any} f nil          = ()
iter {a:Any} f (cons x xs) = f x; iter {a} f xs
```

Explicit type abstractions and type applications in braces.

Two possible polymorphic types for handling state

```
state  : ∀(a:Abs).[](<State Int>(1 → a) → Int → a)
state’ : ∀(a:Any).[](<State Int>(1 → a) → Int → <State Int>a)
```

- `∀(a:Abs)` ascribes kind `Abs` to `a`, allowing values of type `a` to escape the handler.
- `∀(a:Any)` ascribes kind `Any` to `a`, not allowing values of type `a` to escape the handler.

Type polymorphism

Polymorphic version of `iter`

```
iter : ∀(a:Any).[]((a → 1) → List a → 1)
iter {a:Any} f nil          = ()
iter {a:Any} f (cons x xs) = f x; iter {a} f xs
```

Explicit type abstractions and type applications in braces.

Two possible polymorphic types for handling state

```
state  : ∀(a:Abs).[](<State Int>(1 → a) → Int → a)
state’ : ∀(a:Any).[](<State Int>(1 → a) → Int → <State Int>a)
```

- `∀(a:Abs)` ascribes kind `Abs` to `a`, allowing values of type `a` to escape the handler.
- `∀(a:Any)` ascribes kind `Any` to `a`, not allowing values of type `a` to escape the handler.

Using η-expansion we can coerce `state’` to have the type of `state`

```
⊢fun {a:Abs} m s → state’ {a} m s : ∀(a:Abs).[](<State Int>(1 → a) → Int → a) @ .
```

Applying a modality to an absolute type

Modalities act only on non-absolute types, so a modality applied to an absolute type can always be discarded.

Applying a modality to an absolute type

Modalities act only on non-absolute types, so a modality applied to an absolute type can always be discarded.

Examples:

```
⊢ fun x → x : [Gen Int]List Int → List Int @ .
⊬ fun x → x : [Gen Int](1 → 1) → (1 → 1) @ .
a:Any ⊢ fun x → x : <State Int>([Gen Int]a) → [Gen Int]a @ .
a:Any ⊬ fun x → x : <State Int>a → a @ .
a:Abs ⊢ fun x → x : <State Int>a → a @ .
```

The kind restriction on effects

Operation arguments and results are restricted to be absolute.

The kind restriction on effects

Operation arguments and results are restricted to be absolute.

If we allowed `leak:(1 → 1) ↠ 1`, then we could write the following program

```
handle asList (fun () → do leak (fun () → do yield 42)) with
  return _  ⇒  fun () ⇒ 37
  leak p _  ⇒  p
```

which leaks the `yield` operation

The kind restriction on effects

Operation arguments and results are restricted to be absolute.

If we allowed `leak:(1 → 1) ↠ 1`, then we could write the following program

```
handle asList (fun () → do leak (fun () → do yield 42)) with
  return _  ⇒   fun () ⇒ 37
  leak p _  ⇒   p
```

which leaks the `yield` operation

Remark: it is possible to replace this restriction with an alternative formulation in which the order of higher-order effects is important.

Effect pollution

Read and fail effects

```
eff Read = ask : 1 → Int
eff Fail = fail : 1 → 0
```

Effect pollution

Read and fail effects

```
eff Read = ask : 1 → Int
eff Fail = fail : 1 → 0
```

Handling reading from a list of integers (if the list is empty then reading fails):

```
reads : [Fail](<Read>(1 → Int) → List Int → Int)
reads f =
  handle f () with
    return v  ⇒  fun ns → v
    ask () r  ⇒  fun ns → case ns of
                   nil       ⇒  do fail ()
                   cons n ns ⇒  r n ns
```

Effect pollution

Read and fail effects

```
eff Read = ask : 1 → Int
eff Fail = fail : 1 → 0
```

Handling reading from a list of integers (if the list is empty then reading fails):

```
reads : [Fail](<Read>(1 → Int) → List Int → Int)
reads f =
  handle f () with
    return v  ⇒  fun ns → v
    ask () r  ⇒  fun ns → case ns of
                   nil       ⇒  do fail ()
                   cons n ns ⇒  r n ns
```

Handling failure as an option type:

```
maybeFail : [](<Fail>(1 → Int) → Maybe Int)
maybeFail f =
  handle f () with
    return v   ⇒  Just v
    fail () _  ⇒  Nothing
```

Effect pollution

Naively composing `reads` with `maybeFail` leaks the `Fail` effect:

```
bad : [](List Int → <Read, Fail>(1 → Int))
bad ns f = maybeFail (reads f ns)

bad [1,2] (fun () → (do ask ()) + (do fail ())) : Maybe Int @ .
```

This expression evaluates to `Nothing`.

Effect pollution

Naively composing `reads` with `maybeFail` leaks the `Fail` effect:

```
bad : [](List Int → <Read, Fail>(1 → Int))
bad ns f = maybeFail (reads f ns)

bad [1,2] (fun () → (do ask ()) + (do fail ())) : Maybe Int @ .
```

This expression evaluates to `Nothing`.

How can we **encapsulate** the use of `Fail` as an **intermediate** effect?

Effect pollution

Naively composing `reads` with `maybeFail` leaks the `Fail` effect:

```
bad : [](List Int → <Read, Fail>(1 → Int))
bad ns f = maybeFail (reads f ns)

bad [1,2] (fun () → (do ask ()) + (do fail ())) : Maybe Int @ .
```

This expression evaluates to `Nothing`.

How can we **encapsulate** the use of `Fail` as an **intermediate** effect?

The aim is to define

```
good : [](List Int → <Read>(1 → Int) → Maybe Int)
```

by composing `reads` and `maybeFail` such that

```
good [1,2] (fun () → (do ask ()) + (do fail ())) : Maybe Int @ Fail
```

performs the `fail` operation.

Effect encapsulation with masking

The solution is to **mask** the intermediate effect:

```
good : [](List Int → <Read>(1 → Int) → Maybe Int)
good ns f = maybeFail (reads (mask<fail> (f ())))
```

The expression `mask<fail>(M)` masks `fail` from the ambient effect context for `M`.

Effect encapsulation with masking

The solution is to **mask** the intermediate effect:

```
good : [](List Int → <Read>(1 → Int) → Maybe Int)
good ns f = maybeFail (reads (mask<fail> (f ())))
```

The expression `mask<fail>(M)` masks `fail` from the ambient effect context for `M`.

General form `<L|D>` specifies a transformation on effect contexts where:

- `L` is a row of effect labels that are removed from the effect context
- `D` is a row of effects that are added to the effect context

Effect encapsulation with masking

The solution is to **mask** the intermediate effect:

```
good : [](List Int → <Read>(1 → Int) → Maybe Int)
good ns f = maybeFail (reads (mask<fail> (f ())))
```

The expression `mask<fail>(M)` masks `fail` from the ambient effect context for `M`.

General form `<L|D>` specifies a transformation on effect contexts where:

- `L` is a row of effect labels that are removed from the effect context
- `D` is a row of effects that are added to the effect context

`<D>` is shorthand for `<|D>`

Effect polymorphism

Higher-order cooperative concurrency effect

```
eff Coop = fork:[Coop](1 → 1) ↠ 1, suspend:1 ↠ 1
```

But the argument type of `fork` is absolute so cannot support other effects!

Effect polymorphism

Higher-order cooperative concurrency effect

```
eff Coop = fork:[Coop](1 → 1) ↠ 1, suspend:1 ↠ 1
```

But the argument type of `fork` is absolute so cannot support other effects!

METL includes effect polymorphism to support higher-order operations like `fork`

```
eff Coop e = fork:[Coop e, e](1 → 1) ↠ 1, suspend:1 ↠ 1
```

Effect polymorphism

Higher-order cooperative concurrency effect

```
eff Coop = fork:[Coop](1 → 1) ↠ 1, suspend:1 ↠ 1
```

But the argument type of `fork` is absolute so cannot support other effects!

METL includes effect polymorphism to support higher-order operations like `fork`

```
eff Coop e = fork:[Coop e, e](1 → 1) ↠ 1, suspend:1 ↠ 1
```

Effect variables are **only needed** for use-cases such as higher-order effects where a computation must be stored for use in an effect context different from the ambient one.

In the paper (to appear at OOPSLA 2025)

Modal effect types — https://arxiv.org/abs/2407.11816

Met

- simply-typed multimodal core calculus with effects
- type system, operational semantics, type soundness, effect safety
- extensions: sums and products (crisp elimination), type and effect polymorphism

$\mathrm{F}^1_{\mathrm{eff}}$

- restricted core calculus of polymorphic effect types
- restriction: each scope can only refer to the lexically closest effect variables
- encoding of $\mathrm{F}^1_{\mathrm{eff}}$ in Met

Metl: simple bidirectional type checking for Met

- infers all introduction and elimination of modalities
- analogous to generalisation and instantiation

Ongoing and future work

Denotational semantics

Prototype implementation of METL

Extension of MET to support named handlers

Improved (bidirectional) type inference

Combination with oxidizing OCaml (other modalities)