Modal effect types

Sam Lindley

The University of Edinburgh

WG 2.11 Meeting #24, Edinburgh, December 2024

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

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

We can only pass pure functions to `map`

Effect polymorphism

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

An effect-polymorphic version

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

Is this really necessary?

Effect polymorphism

An effect-polymorphic version

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

Is this really necessary?

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

Effect polymorphism

An effect-polymorphic version

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

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 `yield` effect and a generator iterating over a list.

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

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

Effect polymorphism

But Frank's syntactic sugar is fragile.

For instance, consider a `yield` effect and a generator iterating over a list.

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

$$\texttt{gen xs = map (fun x} \rightarrow \texttt{do yield x) xs; ()}$$

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

```
cannot unify effects e and yield, £
```

Effect polymorphism

But Frank's syntactic sugar is fragile.

For instance, consider a `yield` effect and a generator iterating over a list.

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

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

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

```
cannot unify effects e and yield, £
```

Can we do better?

From function arrows to effect contexts

Conventional effect typing — function arrows are annotated with effects

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

From function arrows to effect contexts

Conventional effect typing — function arrows are annotated with effects

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

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

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

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

Modes are effect contexts
Modalities are transformations on modes

Modal effect types

Modes are effect contexts
Modalities are transformations on modes

MET — simply-typed core calculus of modal effect types

Modal effect types

Modes are effect contexts
Modalities are transformations on modes

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

Modes are effect contexts
Modalities are transformations on modes

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

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

Overriding the ambient context with absolute modalities

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

The **absolute modality** `[yield : Int ↠ 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

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

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`.

Overriding the ambient context with absolute modalities

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

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

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`.

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

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 ())
```

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?

`[]((1 → 1) → List Int)` ?

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?

`[]((1 → 1) → List Int)` ?

Unsound as it would allow us to write:

```
mismatch : [Gen String](String → List Int)
mismatch s = asList (fun () → do yield s)
```

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?

`[]((1 → 1) → List Int)` ?

Unsound as it would allow us to write:

```
mismatch : [Gen String](String → List Int)
mismatch s = asList (fun () → do yield s)
```

`String` handled as `Int`!

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 a)` ?

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 a)` ?

Sound, but consider:

$\vdash$ `fun` $\underbrace{\texttt{f}}_{\texttt{@ Gen Int}}$ $\rightarrow$ `handle` $\underbrace{\texttt{f ()}}_{\texttt{@ Gen Int, E}}$ `with asList f : [Gen Int]`$(\underbrace{\texttt{1} \rightarrow \texttt{1}}_{\texttt{@ Gen Int}})$ $\rightarrow$ `List 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?

`[]([Gen Int](1 → 1) → List a)` ?

Sound, but consider:

`⊢ fun f → handle f () with asList f : [Gen Int](1 → 1) → List Int @ E`

(where `f` is `@ Gen Int`, `f ()` is `@ Gen Int, E`, and `1 → 1` is `@ Gen Int`)

Restriction to `[Gen Int]` severely hinders resuability

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 a)
```

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

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

Now 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?

```
[](<Gen Int>(1 → 1) → List a)
```

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

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

Now the effect context of `f` is `Gen Int, E`.

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

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

Coercions between modalities

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

We can extend an absolute modality:

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

Coercions between modalities

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

We can extend an absolute modality:

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

We cannot extend a relative modality in the same way:

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

Coercions between modalities

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

We can extend an absolute modality:

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

We cannot extend a relative modality in the same way:

```
⊬ fun f → f : <>(1 → 1) → <Gen Int>( 1 → 1 ) @ E  # Ill-typed
                 └@ E┘               └@ Gen Int, E┘
```

This would insert a fresh `yield : Int ↠ 1` operation which may shadow other `yield` operations in `E`, permitting bad programs like `mismatch`.

Coercions between modalities

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

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

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
                       @ Gen Int             @ Gen Int, E
```

But the converse is not permitted

```
⊬ fun f → f : <Gen Int>( 1 → 1 ) → [Gen Int](1 → 1) @ E  # Ill-typed
                       @ Gen Int, E          @ Gen Int
```

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

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

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

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

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

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

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

Masking

Using a generator to find an integer satisfying a predicate

```
findWrong : []((Int → Bool) → List Int → Maybe Int)  # ill-typed
findWrong p xs =
  handle (iter (fun x → if p x then do yield x) xs) with
    return _  ⇒ nothing
    yield x _ ⇒ just x
```

Masking

Using a generator to find an integer satisfying a predicate

```
findWrong : []((Int → Bool) → List Int → Maybe Int)  # ill-typed
findWrong p xs =
  handle (iter (fun x → if p x then do yield x) xs) with
    return _   ⇒ nothing
    yield x _ ⇒ just x
```

Unsound to invoke `p` in the scope of the handler — would accidentally handle any yield operations performed by `p`

```
⊢ ... handle (iter (fun x → if (p x) then do yield x) xs) with ...  :  _  @  E
                                \___/
                            @ Gen Int, E
```

Masking

Using a generator to find an integer satisfying a predicate

```
findWrong : []((Int → Bool) → List Int → Maybe Int)  # ill-typed
findWrong p xs =
  handle (iter (fun x → if p x then do yield x) xs) with
    return _  ⇒ nothing
    yield x _ ⇒ just x
```

Unsound to invoke `p` in the scope of the handler — would accidentally handle any yield operations performed by `p`

```
⊢ ... handle (iter (fun x → if (p x) then do yield x) xs) with ...  :  _  @  E
                               ⏟
                          @ Gen Int, E
```

Changing the type of `p` to `<Gen Int>(Int → Bool)` fixes the type error but leaks the implementation detail that `findWrong` uses `yield`

Masking

Using a generator to find an integer satisfying a predicate

```
findWrong : []((Int → Bool) → List Int → Maybe Int)  # ill-typed
findWrong p xs =
  handle (iter (fun x → if p x then do yield x) xs) with
    return _  ⇒ nothing
    yield x _ ⇒ just x
```

Unsound to invoke `p` in the scope of the handler — would accidentally handle any yield operations performed by `p`

```
⊢ ... handle (iter (fun x → if (p x) then do yield x) xs) with ...  : _ @ E
                               └─ @ Gen Int, E
```

Changing the type of `p` to `<Gen Int>(Int → Bool)` fixes the type error but leaks the implementation detail that `findWrong` uses `yield`

Masking solves the problem

```
⊢ ... handle (iter (fun x → if mask<yield>(p x) ... ) with ...  : _ @ E
                                          └─ @ E
```

Masking

`mask<yield>(M)` **masks** `yield` from the ambient effect context for `M`.

Masking

`mask<yield>(M)` **masks** `yield` from the ambient effect context for `M`.

`mask<yield>(p x)` initially returns a value of type `<yield|>Bool` instead of `Bool`, where `<yield|>` is a relative modality masking `yield` from the ambient effect context.

Masking

mask<yield>(M) **masks** yield from the ambient effect context for M.

mask<yield>(p x) initially returns a value of type <yield|>Bool instead of Bool, where <yield|> is a relative modality masking yield from the ambient effect context.

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

Masking

`mask<yield>(M)` **masks** `yield` from the ambient effect context for `M`.

`mask<yield>(p x)` initially returns a value of type `<yield|>Bool` instead of `Bool`, where `<yield|>` is a relative modality masking `yield` from the ambient effect context.

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>`

Kinds

State handler for `1` $\to$ `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** formed only from base types and types boxed by an absolute modality cannot leak effects
- **Unrestricted types** may include functions not boxed by an absolute modality so may leak effects

Kinds

- **Absolute types** formed only from base types and types boxed by an absolute modality cannot leak effects
- **Unrestricted types** may include functions not boxed by an absolute modality so may leak effects

Kinds:

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

Kinds

- **Absolute types** formed only from base types and types boxed by an absolute modality cannot leak effects
- **Unrestricted types** may include functions not boxed by an absolute modality so may leak effects

Kinds:

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

Subkinding allows absolute types to be treated as unrestricted

Type polymorphism

Polymorphic version of `iter`

```
iter : ∀ a . [](( a → 1) → List a → 1)
iter {a} f nil          = ()
iter {a} 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 . []((a → 1) → List a → 1)
iter {a} f nil          = ()
iter {a} 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] . [](<State Int>(1 → a) → Int → a)
state' : ∀  a  . [](<State Int>(1 → a) → Int → <State Int>a)
```

- `∀ [a]` ascribes kind `Abs` to `a`, allowing values of type `a` to escape the handler.
- `∀ a` ascribes kind `Any` to `a`, not allowing values of type `a` to escape the handler.

Type polymorphism

Polymorphic version of `iter`

```
iter : ∀ a . []((a → 1) → List a → 1)
iter {a} f nil          = ()
iter {a} 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] . []( <State Int>(1 → a) → Int → a)
state' : ∀  a  . []( <State Int>(1 → a) → Int → <State Int>a)
```

- `∀ [a]` ascribes kind `Abs` to `a`, allowing values of type `a` to escape the handler.
- `∀ a` 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} m s → state' {a} m s : ∀ [a] . []( <State Int>(1 → a) → Int → 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

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

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