Modal effect types

Sam Lindley

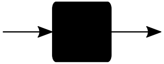
The University of Edinburgh

Bristol, 29th January 2026

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

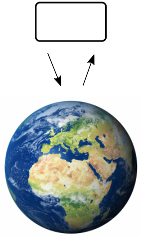
Effects

Programs as black boxes (Church-Turing model)?



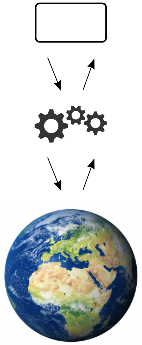
Effects

Programs must interact with their environment



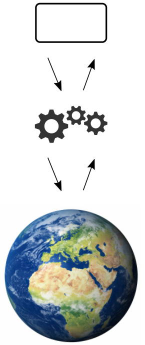
Effects

Programs must interact with their environment



Effects

Programs must interact with their environment

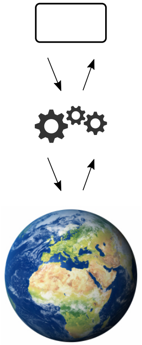


Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

Effects

Programs must interact with their environment



Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

Effect type systems statically track the use of effects

Conventional effect types

Pure computation

```
inc : Int → Int
inc i = i + 1

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

```
> app inc 42
43 : Int
```

Conventional effect types

A variant of `inc` using a `Read` effect supporting effectful operation `ask : 1 ↠ Int`

```
inc : Int → Int
inc i = i + do ask ()
```

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

Conventional effect types

Effects are tracked statically by adding effect annotations to arrows

```
inc : Int --Read--> Int
inc i = i + do ask ()

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

Conventional effect types

Effect polymorphism allows `app` to be used in the presence of arbitrary effects

$$\texttt{inc} : \texttt{Int} \xrightarrow{\texttt{Read}} \texttt{Int}$$

```
inc i = i + do ask ()
```

$$\texttt{app} : \forall\, \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$$

```
app f x = f x
```

$$\texttt{appinc} : \texttt{Int} \xrightarrow{\texttt{Read}} \texttt{Int}$$

```
appinc = app inc
```

Conventional effect types

Effect polymorphism also allows `inc` to be used in contexts that have additional effects

$$\texttt{inc} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, e}} \texttt{Int}$$

```
inc i = i + do ask ()
```

$$\texttt{app} : \forall\ \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$$

```
app f x = f x
```

$$\texttt{inp} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, IO, e}} \texttt{Int}$$

```
inp i = do print "incrementing"; inc i
```

Conventional effect types

Effect polymorphism tracks a handler consuming an effect

$$\texttt{inc} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, e}} \texttt{Int}$$

```
inc i = i + do ask ()
```

$$\texttt{app} : \forall\ \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$$

```
app f x = f x
```

$$\texttt{two} : \forall\ \texttt{e}.(1 \xrightarrow{\texttt{Read, e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int}$$

```
two f = handle f () with {ask () r ⇒ r 2}
```

```
> two (fun () → app inc 42)
44 : Int
```

Conventional effect types

$$\texttt{inc} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, e}} \texttt{Int}$$

$$\texttt{inp} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, IO, e}} \texttt{Int}$$

$$\texttt{app} : \forall\ \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$$

$$\texttt{two} : \forall\ \texttt{e}.(\texttt{1} \xrightarrow{\texttt{Read, e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int}$$

Do we really need all of these effect variables?

Can we add effect types to existing languages without having to rewrite signatures of higher-order functions such `app`?

Modal effect types

Key ideas

- **decouple** effect types from function arrows
- track effects through an **ambient effect context**
- use **modalities** to modify the ambient effect context locally

Modal effect types

Pure computation

```
inc : Int → Int
inc i = i + 1

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

```
> app inc 42
43 : Int
```

Modal effect types

A variant of `inc` using a `Read` effect supporting effectful operation `ask : 1 ↠ Int`

```
inc : Int → Int
inc i = i + do ask ()
```

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

Modal effect types

Effects (and purity) are tracked statically using **absolute modalities**

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

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

Modal effect types

Subeffecting allows `app` to be used in the presence of arbitary effects

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

app : []((Int → Int) → Int → Int)
app f x = f x

appinc : [Read](Int → Int)
appinc = app inc
```

Modal effect types

Subeffecting also allows `inc` to be used in contexts that have other effects too

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

app : []((Int → Int) → Int → Int)
app f x = f x

inp : [Read, IO](Int → Int)
inp i = do print "incrementing"; inc i
```

Modal effect types

Relative modalities track a handler consuming an effect

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

app : []((Int → Int) → Int → Int)
app f x = f x

two : []((<Read>(1 → Int)) → Int)
two f = handle f () with {ask () r ⇒ r 2}

> two (fun () → app inc 42)
44 : Int
```

Comparing conventional effect types with modal effect types

Conventional effect types

$$\mathtt{inc} : \forall\ \mathtt{e}.\mathtt{Int} \xrightarrow{\mathtt{Read,\ e}} \mathtt{Int}$$

$$\mathtt{inp} : \forall\ \mathtt{e}.\mathtt{Int} \xrightarrow{\mathtt{Read,\ IO,\ e}} \mathtt{Int}$$

$$\mathtt{app} : \forall\ \mathtt{e}.(\mathtt{Int} \xrightarrow{\mathtt{e}} \mathtt{Int}) \xrightarrow{\mathtt{e}} \mathtt{Int} \xrightarrow{\mathtt{e}} \mathtt{Int}$$

$$\mathtt{two} : \forall\ \mathtt{e}.(\mathtt{1} \xrightarrow{\mathtt{Read,\ e}} \mathtt{Int}) \xrightarrow{\mathtt{e}} \mathtt{Int}$$

Modal effect types

$$\mathtt{inc} : [\mathtt{Read}](\mathtt{Int} \to \mathtt{Int})$$

$$\mathtt{inp} : [\mathtt{Read},\ \mathtt{IO}](\mathtt{Int} \to \mathtt{Int})$$

$$\mathtt{app} : [\,]((\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \to \mathtt{Int})$$

$$\mathtt{two} : [\,](\langle\mathtt{Read}\rangle(\mathtt{1} \to \mathtt{Int}) \to \mathtt{Int})$$

Comparing conventional effect types with modal effect types

Conventional effect types

$$\texttt{inc} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, e}} \texttt{Int}$$
$$\texttt{inp} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, IO, e}} \texttt{Int}$$
$$\texttt{app} : \forall\ \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$$
$$\texttt{two} : \forall\ \texttt{e}.(\texttt{1} \xrightarrow{\texttt{Read, e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int}$$

Modal effect types

$$\texttt{inc} : [\texttt{Read}](\texttt{Int} \rightarrow \texttt{Int})$$
$$\texttt{inp} : [\texttt{Read, IO}](\texttt{Int} \rightarrow \texttt{Int})$$
$$\texttt{app} : (\texttt{Int} \rightarrow \texttt{Int}) \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$
$$\texttt{two} : \langle\texttt{Read}\rangle(\texttt{1} \rightarrow \texttt{Int}) \rightarrow \texttt{Int}$$

(we allow top-level empty absolute modalities to be omitted)

Comparing conventional effect types with modal effect types

Conventional effect types

$\texttt{inc} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, e}} \texttt{Int}$

$\texttt{inp} : \forall\ \texttt{e}.\texttt{Int} \xrightarrow{\texttt{Read, IO, e}} \texttt{Int}$

$\texttt{app} : \forall\ \texttt{e}.(\texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int} \xrightarrow{\texttt{e}} \texttt{Int}$

$\texttt{two} : \forall\ \texttt{e}.(\texttt{1} \xrightarrow{\texttt{Read, e}} \texttt{Int}) \xrightarrow{\texttt{e}} \texttt{Int}$

Modal effect types

$\texttt{inc} : [\texttt{Read}](\texttt{Int} \to \texttt{Int})$

$\texttt{inp} : [\texttt{Read, IO}](\texttt{Int} \to \texttt{Int})$

$\texttt{app} : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}$

$\texttt{two} : \langle\texttt{Read}\rangle(\texttt{1} \to \texttt{Int}) \to \texttt{Int}$

Modal effect types allow us to avoid unnecessary effect polymorphism

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

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

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

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

Effect contexts

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

Effect contexts

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

Example: `ask:1` $\twoheadrightarrow$ `Int`, `print:String` $\twoheadrightarrow$ `1`

Effect contexts

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

Example: `ask`:`1` $\twoheadrightarrow$ `Int`, `print`:`String` $\twoheadrightarrow$ `1`

For convenience, we often group related operations together as effects

Effect contexts

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

Example: `ask:1 ↠ Int, print:String ↠ 1`

For convenience, we often group related operations together as effects

Examples:

```
eff Read = ask:1 ↠ Int
eff IO = print:String ↠ 1
eff State a = get:1 ↠ a, put:a ↠ 1
eff Gen a = yield:a ↠ 1
```

Effect contexts

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

Example: `ask:1 ↠ Int, print:String ↠ 1`

For convenience, we often group related operations together as effects

Examples:

```
eff Read = ask:1 ↠ Int
eff IO = print:String ↠ 1
eff State a = get:1 ↠ a, put:a ↠ 1
eff Gen a = yield:a ↠ 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 typing

A **mode** is an effect context

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

Modal effect typing

A **mode** is an effect context

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

$\textsc{Met}$ — simply-typed core calculus of modal effect types

Modal effect typing

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 typing

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

$$\vdash \ \underbrace{\texttt{fun x} \to \texttt{do yield (x + 42)}}_{}\ $$

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

Overriding the ambient context with absolute modalities

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

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

Overriding the ambient context with absolute modalities

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

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

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

Overriding the ambient context with absolute modalities

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

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

The **absolute modality** `[Gen Int]` **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

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

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

The **absolute modality** `[Gen Int]` **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

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

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

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

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

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

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

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

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

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

A flavour of the typing rules (courtesy of Wenhao Tang)

A Tale of Locks 🔒 and Keys 🔑

mod introduces a modality and a lock

the ambient effect context is overwritten to ask

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

locks control usage of variables

modality transformation: the key 🔑 to the lock

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

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

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

Coercions between modalities

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

We can extend an absolute modality:

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

Coercions between modalities

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

We can extend an absolute modality:

$\vdash$ `fun f -> f : [Gen Int](1 -> 1) -> [Gen Int, Gen String](1 -> 1) @ E`

In a conventional effect type system this corresponds to:

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

Coercions between modalities

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

We can extend an absolute modality:

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

In a conventional effect type system this corresponds to:

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

We cannot extend a relative modality in the same way:

$\nvdash$ `fun f → f : <>(1 → 1) → <Gen Int>(1 → 1) @ E  # Ill-typed`

Coercions between modalities

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

We can extend an absolute modality:

$\vdash$ `fun f -> f : [Gen Int](1 -> 1) -> [Gen Int, Gen String](1 -> 1) @ 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:

$\nvdash$ `fun f -> f : <>(1 -> 1) -> <Gen Int>(1 -> 1) @ E  # Ill-typed`

This would insert a fresh `yield:Int` $\twoheadrightarrow$ `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:

$\vdash$ `fun f -> f : [Gen Int](1 -> 1) -> [Gen Int, Gen String](1 -> 1) @ 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:

$\nvdash$ `fun f -> f : <>(1 -> 1) -> <Gen Int>(1 -> 1) @ E  # Ill-typed`

This would insert a fresh `yield:Int ->> 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.

$\vdash$ `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.

$\vdash$ `fun f` $\rightarrow$ `f : [Gen Int](1` $\rightarrow$ `1)` $\rightarrow$ `<Gen Int>(1` $\rightarrow$ `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.

$\vdash$ `fun f -> f : [Gen Int](1 -> 1) -> <Gen Int>(1 -> 1) @ E`

In a conventional effect type system this corresponds to:

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

But the converse is not permitted

$\nvdash$ `fun f -> f : <Gen Int>(1 -> 1) -> [Gen Int](1 -> 1) @ E  # 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.

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

In a conventional effect type system this corresponds to:

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

But the converse is not permitted

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

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

In a conventional effect type system this corresponds to:

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

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

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

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

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

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

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

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** (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` $\to$ `1)`)
  built from base types, positive types, and types boxed by an absolute modality —
  cannot leak effects
- **Unrestricted types** (e.g. `1` $\to$ `1`, `List Int` $\to$ `1`, and `<Coop>(1` $\to$ `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 (OOPSLA 2025)

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

$\textsc{Met}$

- 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 $\textsc{Met}$

$\textsc{Metl}$: simple bidirectional type checking for $\textsc{Met}$

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

In the follow-up paper (POPL 2026)

Rows and capabilities as modal effects (Wenhao Tang and Sam Lindley)

https://arxiv.org/abs/2507.10301

$\textsc{Met}(\mathcal{X})$

- abstracts over **effect structure** $\mathcal{X}$
- first class labels and modality parameterised handlers
- encodings of core calculi of Koka and Effekt in $\textsc{Met}(\mathcal{X})$

Ongoing and future work

Denotational semantics

Prototype implementation of METL

Improved (bidirectional) type inference — Frost

Combination with oxidizing OCaml (other modalities)

Inspirations

Do be do be do. *Lindley, McBride, and McLaughlin*. POPL 2017

Doo bee doo bee doo. *Convent, Lindley, McBride, and McLaughlin*. JFP 2020

Effekt. *Brachthäuser, Schuster, and Ostermann*

Oxidizing OCaml. *Lorenzen, White, Dolan, Eisenberg, and Lindley*. ICFP 2024

Multimodal dependent type theory. *Gratzer, Kavvos, Nuyts, and Birkedal*. LMCS 2021