Rows and Capabilities as Modal Effects

Wenhao Tang [1] Sam Lindley [1]

HOPE workshop, Singapore, 12th Oct 2025

[1]

THE UNIVERSITY *of* EDINBURGH

Effects and Effect Types

Effects are the way programs interact with their environment

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

Effect types statically tracks use of effects

Many recent practical effect systems as based on rows or capabilities.

Row-Based Effect Types

Row-Based Effect Types à la Koka

System F^{ϵ} formalises Koka's row-based effect system[1]

Key idea: annotate each function arrow with a row of effects

$$A \to^{E} B$$

A function that may invokes effects in E when applied

[1] Xie, Brachthäuser, Hillerström, Schuster, and Leijen, "Effect handlers, evidently", 2020.

A First-Order Effectful Function

An operation $\texttt{yield} : \texttt{Int} \Rightarrow \texttt{1}$

$$\lambda^{\texttt{yield}} x^{\texttt{Int}}.\texttt{do}\ \texttt{yield}\ x \ : \ \texttt{Int} \longrightarrow^{\texttt{yield}} \texttt{1}$$

A First-Order Effectful Function

An operation $\texttt{yield} : \texttt{Int} \Rightarrow \texttt{1}$

$$\lambda^{\texttt{yield}} x^{\texttt{Int}}.\texttt{do yield}\ x \ : \ \texttt{Int} \to^{\texttt{yield}} \texttt{1}$$

Typing derivation

$$\textsc{T-Abs}\ \frac{\textsc{T-Do}\ \dfrac{\texttt{yield} : \texttt{Int} \Rightarrow \texttt{1} \qquad \Gamma, x : \texttt{Int} \vdash x \ : \ \texttt{Int}}{\Gamma, x : \texttt{Int} \vdash \texttt{do yield}\ x \ : \ \texttt{1} \mid \texttt{yield}}}{\Gamma \vdash \lambda^{\texttt{yield}} x^{\texttt{Int}}.\texttt{do yield}\ x \ : \ \texttt{Int} \to^{\texttt{yield}} \texttt{1}}$$

Functions are pure; when creating a function, we must track the effects they may use when applied in its type.

Parametric Effect Polymorphism

A higher-order application function

$$\lambda f^{\mathtt{Int} \to^E \mathtt{1}}.\lambda^E x^{\mathtt{Int}}.f\ x \ : \ (\mathtt{Int} \to^E \mathtt{1}) \to \mathtt{Int} \to^E \mathtt{1}$$

Parametric Effect Polymorphism

A higher-order application function

$$\lambda f^{\mathtt{Int} \to^{E} \mathtt{1}}.\lambda^{E} x^{\mathtt{Int}}.f\ x \ : \ (\mathtt{Int} \to^{E} \mathtt{1}) \to \mathtt{Int} \to^{E} \mathtt{1}$$

Abstract over the effect type E via an effect variable ε

$$\Lambda\varepsilon.\lambda f^{\mathtt{Int} \to^{\varepsilon} \mathtt{1}}.\lambda^{\varepsilon} x^{\mathtt{Int}}.f\ x \ : \ \forall\varepsilon.(\mathtt{Int} \to^{\varepsilon} \mathtt{1}) \to \mathtt{Int} \to^{\varepsilon} \mathtt{1}$$

Capability-Based Effect Types

Capability-Based Effect Types à la Effekt

System C formalises Effekt's capability-based effect system[2]

Key idea: treat effects as capabilities provided by the context

$$(\overline{A}, \overline{f : T}) \Rightarrow B$$

A block (i.e., second-class function) binds capabilities $\overline{f : T}$ and may use them as well as other capabilities from the context

[2] Brachthäuser, Schuster, Lee, and Boruch-Gruszecki, "Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back", 2022.

Blocks

System C distinguishes between first-class values and second-class blocks (i.e., functions)

A higher-order block

$$app_C \doteq \{(x : \texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow f(x)\} \;:\; (\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1}$$

- the second-class block parameter $f : \texttt{Int} \Rightarrow \texttt{1}$ is a capability
- blocks must be fully applied and cannot be passed/stored as values
- the block body can use capabilities from the context

$$y :^{*} \texttt{Int} \Rightarrow \texttt{1} \vdash \{(x : \texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow y(x)\} \;:\; (\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1}$$

Boxing Blocks into First-Class Values

In order to define a curried app_C we need to turn blocks into first-class values

$$app'_C \doteq \{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \text{box } \{(x : \texttt{Int}) \Rightarrow f(x)\}\} \; : \; (f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1} \text{ at } \{f\})$$

Boxing Blocks into First-Class Values

In order to define a curried app_C we need to turn blocks into first-class values

$$app'_C \doteq \{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \textsf{box}\ \{(x : \texttt{Int}) \Rightarrow f(x)\}\} \ : \ (f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1}\ \textsf{at}\ \{f\})$$

- box $\cdots$ turns a block into a first-class value

Boxing Blocks into First-Class Values

In order to define a curried app_C we need to turn blocks into first-class values

$$app'_C \doteq \{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \textsf{box}\ \{(x : \texttt{Int}) \Rightarrow f(x)\}\} \ : \ (f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1}\ \textsf{at}\ \{f\})$$

- box $\cdots$ turns a block into a first-class value
- the result type $\texttt{Int} \Rightarrow \texttt{1}$ at $\{f\}$ tracks that this boxed block uses the capability f

Unifying / Comparing Rows and Capabilities?

Yoshioka et al.[3] unify different row-based effect systems

It is non-obvious how to systematically translate between System F$^\epsilon$ and System C

$$\text{System F}^\epsilon : A \to^{E} B$$

$$\text{System C} : (\overline{A}, \overline{f : T}) \Rightarrow B$$

The main problem: effect tracking is deeply **entangled** with other type system features such as function types

[3] Yoshioka, Sekiyama, and Igarashi, “Abstracting Effect Systems for Algebraic Effect Handlers”, 2024.

A Detour: CBV, CBN, and CBPV

Translating between CBV and CBN is rather involved ...

A Detour: CBV, CBN, and CBPV

Translating between CBV and CBN is rather involved ...

... because when to suspend and force computations is **entangled** with functions

A Detour: CBV, CBN, and CBPV

Translating between CBV and CBN is rather involved …

… because when to suspend and force computations is **entangled** with functions

CBPV smoothly subsumes both by **decoupling** thunking and forcing from function abstraction and application

A Detour: CBV, CBN, and CBPV

Translating between CBV and CBN is rather involved …

… because when to suspend and force computations is **entangled** with functions

CBPV smoothly subsumes both by **decoupling** thunking and forcing from function abstraction and application

Why not decouple effect tracking from function types?

Modal Effect Types

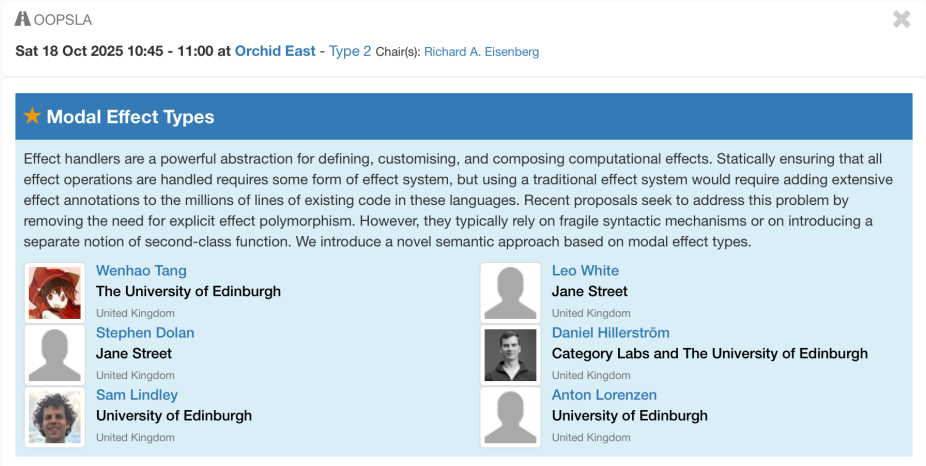
Modal Effect Types

Modal effect types (MET) is a recent effect system based on Multimodal Type Theory[4] and inspired by languages Frank[5] and Effekt[6]

[4] Gratzer, Kavvos, Nuyts, and Birkedal, "Multimodal Dependent Type Theory", 2020.
[5] Lindley, McBride, and McLaughlin, "Do Be Do Be Do", 2017.
[6] Brachthäuser, Schuster, and Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism", 2020.

Modal Effect Types

Modal effect types (MET) is a recent effect system based on Multimodal Type Theory[4] and inspired by languages Frank[5] and Effekt[6]


OOPSLA

Sat 18 Oct 2025 10:45 - 11:00 at Orchid East - Type 2 Chair(s): Richard A. Eisenberg

★ Modal Effect Types

Effect handlers are a powerful abstraction for defining, customising, and composing computational effects. Statically ensuring that all effect operations are handled requires some form of effect system, but using a traditional effect system would require adding extensive effect annotations to the millions of lines of existing code in these languages. Recent proposals seek to address this problem by removing the need for explicit effect polymorphism. However, they typically rely on fragile syntactic mechanisms or on introducing a separate notion of second-class function. We introduce a novel semantic approach based on modal effect types.

Wenhao Tang
The University of Edinburgh
United Kingdom

Stephen Dolan
Jane Street
United Kingdom

Sam Lindley
University of Edinburgh
United Kingdom

Leo White
Jane Street
United Kingdom

Daniel Hillerström
Category Labs and The University of Edinburgh
United Kingdom

Anton Lorenzen
University of Edinburgh
United Kingdom

Modal Effect Types

Modal effect types (MET) is a recent effect system based on Multimodal Type Theory[4] and inspired by languages Frank[5] and Effekt[6]

MET **decouples** effect tracking from standard type and term constructs via modalities

[4] Gratzer, Kavvos, Nuyts, and Birkedal, "Multimodal Dependent Type Theory", 2020.
[5] Lindley, McBride, and McLaughlin, "Do Be Do Be Do", 2017.
[6] Brachthäuser, Schuster, and Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism", 2020.

Modal Effect Types

Modal effect types (MET) is a recent effect system based on Multimodal Type Theory[4] and inspired by languages Frank[5] and Effekt[6]

MET **decouples** effect tracking from standard type and term constructs via modalities

The decoupling gives us the flexibility and expressivity to compositionally encode different effect tracking mechanisms.

[4] Gratzer, Kavvos, Nuyts, and Birkedal, "Multimodal Dependent Type Theory", 2020.
[5] Lindley, McBride, and McLaughlin, "Do Be Do Be Do", 2017.
[6] Brachthäuser, Schuster, and Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism", 2020.

Modal Effect Types

Modal effect types (MET) is a recent effect system based on Multimodal Type Theory[4] and inspired by languages Frank[5] and Effekt[6]

MET **decouples** effect tracking from standard type and term constructs via modalities

The decoupling gives us the flexibility and expressivity to compositionally encode different effect tracking mechanisms.

Based on MET, we give a uniform framework MET($\mathcal{X}$) for encoding and comparing different effect systems

[4] Gratzer, Kavvos, Nuyts, and Birkedal, "Multimodal Dependent Type Theory", 2020.
[5] Lindley, McBride, and McLaughlin, "Do Be Do Be Do", 2017.
[6] Brachthäuser, Schuster, and Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism", 2020.

Effect Contexts

A typing judgement tracks the **ambient effect context**

$$\vdash \lambda x^{\mathtt{Int}}.\mathsf{do}\ \mathtt{yield}\ x\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}$$

Effect Contexts

A typing judgement tracks the **ambient effect context**

$$\vdash \lambda x^{\mathtt{Int}}.\mathsf{do}\ \mathtt{yield}\ x\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}$$

The whole term shares the ambient effect context

$$\vdash (\lambda f^{\mathtt{Int}\to\mathtt{1}}.\lambda x^{\mathtt{Int}}.f\ x)\ (\lambda x^{\mathtt{Int}}.\mathsf{do}\ \mathtt{yield}\ x)\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}$$

Effect Contexts

A typing judgement tracks the **ambient effect context**

$$\vdash \lambda x^{\mathtt{Int}}.\mathtt{do}\ \mathtt{yield}\ x\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}$$

The whole term shares the ambient effect context

$$\vdash (\lambda f^{\mathtt{Int}\to\mathtt{1}}.\lambda x^{\mathtt{Int}}.f\ x)\ (\lambda x^{\mathtt{Int}}.\mathtt{do}\ \mathtt{yield}\ x)\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}$$

A natural notion of sub-effecting

$$\vdash \lambda x^{\mathtt{Int}}.\mathtt{do}\ \mathtt{yield}\ x\ :\ \mathtt{Int} \longrightarrow \mathtt{1}\ @\ \mathtt{yield}, \mathtt{ask}$$

Modalities

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

$$\frac{🍋_{[\texttt{yield}]} \vdash \lambda x^{\texttt{Int}}.\texttt{do yield } x \ : \ \texttt{Int} \longrightarrow \texttt{1} \ @ \ \texttt{yield}}{\vdash \ \mathsf{mod}_{[\texttt{yield}]} \ (\lambda x^{\texttt{Int}}.\texttt{do yield } x) \ : \ [\texttt{yield}](\texttt{Int} \longrightarrow \texttt{1}) \ @ \ F}$$

$\mathsf{mod}_{[\texttt{yield}]}$ introduces a lock $🍋_{[\texttt{yield}]}$ to the context of the premise [7]

[7] XeLaTeX complains about my lock symbol so I have to use a lemon instead

Modalities

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

$$\frac{🍃_{[\texttt{yield}]} \vdash \lambda x^{\texttt{Int}}.\texttt{do yield } x \ : \ \texttt{Int} \to \texttt{1} \ @ \ \texttt{yield}}{\vdash \mathsf{mod}_{[\texttt{yield}]} \ (\lambda x^{\texttt{Int}}.\texttt{do yield } x) \ : \ [\texttt{yield}](\texttt{Int} \to \texttt{1}) \ @ \ F}$$

$\mathsf{mod}_{[\texttt{yield}]}$ introduces a lock $🍃_{[\texttt{yield}]}$ to the context of the premise [7]

A relative modality $\langle E \rangle$ adds effects E to the ambient effect context ($\langle E \rangle(F) = E, F$)

$$\frac{🍃_{\langle \texttt{yield} \rangle} \vdash \lambda x^{\texttt{Int}}.\texttt{do yield (do ask ())} \ : \ \texttt{Int} \to \texttt{1} \ @ \ \texttt{yield}, \texttt{ask}}{\vdash \mathsf{mod}_{\langle \texttt{yield} \rangle} \ (\lambda x^{\texttt{Int}}.\texttt{do yield (do ask ())}) \ : \ \langle \texttt{yield} \rangle(\texttt{Int} \to \texttt{1}) \ @ \ \texttt{ask}}$$

[7] XeLaTeX complains about my lock symbol so I have to use a lemon instead

Locks

Locks control the accessibility of variables

An invalid judgement

$$f : \texttt{Int} \to \texttt{1} \nvdash \mathsf{mod}_{[\texttt{yield}]}\ (\lambda x^{\texttt{Int}}.f\ x)\ :\ [\texttt{yield}](\texttt{Int} \to \texttt{1})\ @\ \texttt{ask}$$

Locks

Locks control the accessibility of variables

An invalid judgement

$$f : \mathtt{Int} \to \mathtt{1} \nvdash \mathsf{mod}_{[\mathtt{yield}]}\ (\lambda x^{\mathtt{Int}}.f\ x)\ :\ [\mathtt{yield}](\mathtt{Int} \to \mathtt{1})\ @\ \mathtt{ask}$$

Because its expected premise does not hold

$$f : \mathtt{Int} \to \mathtt{1}, \blacksquare_{[\mathtt{yield}]} \nvdash \lambda x^{\mathtt{Int}}.f\ x\ :\ \mathtt{Int} \to \mathtt{1}\ @\ \mathtt{yield}$$

Modality Elimination

We can make the premise well-typed by annotating the binding of f with [] (or [yield])

$$f :_{[]} \texttt{Int} \to \texttt{1}, \text{🍃}_{[\texttt{yield}]} \vdash \lambda x^{\texttt{Int}}.f\ x \ : \ \texttt{Int} \to \texttt{1} \ @ \ \texttt{yield}$$

Modality Elimination

We can make the premise well-typed by annotating the binding of f with [] (or [yield])

$$f :_{[]} \texttt{Int} \to \texttt{1}, \blacksquare_{[\texttt{yield}]} \vdash \lambda x^{\texttt{Int}}.f\ x \ : \ \texttt{Int} \to \texttt{1} \ @ \ \texttt{yield}$$

Such a binding is introduced by modality elimination (the default annotation is ⟨⟩)

$$\frac{\vdash V \ : \ []\texttt{(Int} \to \texttt{1)} \qquad f :_{[]} \texttt{Int} \to \texttt{1} \vdash M \ : \ A \ @ \ \texttt{yield}}{\vdash \textsf{let mod}_{[]}\ f = V \ \textsf{in}\ M \ : \ A \ @ \ \texttt{yield}}$$

Modality Transformations

How does the type system decide that $f :_{[]} \texttt{Int} \longrightarrow \texttt{1}$ can be used after  $_{\texttt{[yield]}}$?

Modality Transformations

How does the type system decide that $f :_{[]} \texttt{Int} \to \texttt{1}$ can be used after $\blacksquare_{[\texttt{yield}]}$?

$$\text{T-Var}\ \frac{[] \Rightarrow [\texttt{yield}]}{f :_{[]} \texttt{Int} \to \texttt{1}, \blacksquare_{[\texttt{yield}]}, x : \texttt{Int} \vdash f\ :\ \texttt{Int} \to \texttt{1}\ @\ \texttt{yield}}$$

Intuitively, $\mu \Rightarrow \nu$ allows us to use a variable $f :_{\mu} A$ after a lock $\blacksquare_{\nu}$

Modality Transformations

How does the type system decide that $f :_{[]} \texttt{Int} \to \texttt{1}$ can be used after $\blacklozenge_{[\texttt{yield}]}$?

$$\textsc{T-Var}\ \frac{[] \Rightarrow [\texttt{yield}]}{f :_{[]} \texttt{Int} \to \texttt{1}, \blacklozenge_{[\texttt{yield}]}, x : \texttt{Int} \vdash f\ :\ \texttt{Int} \to \texttt{1}\ @\ \texttt{yield}}$$

Intuitively, $\mu \Rightarrow \nu$ allows us to use a variable $f :_{\mu} A$ after a lock $\blacklozenge_{\nu}$

Examples:

- $[E] \Rightarrow [F]$ holds if $E \leqslant F$
- $[] \Rightarrow \mu$ holds for any μ
- $\langle\rangle \Rightarrow [E]$ does not hold for any E

Modality Composition

What if there are multiple locks?

$$\text{T-VAR}\ \frac{\mu \Rightarrow \nu_1 \circ \nu_2 \circ \nu_3}{f :_{\mu} A, \blacksquare_{\nu_1}, \blacksquare_{\nu_2}, \blacksquare_{\nu_3} \vdash f \ : \ A \ @ \ E}$$

Modality Composition

What if there are multiple locks?

$$\text{T-VAR}\ \frac{\mu \Rightarrow \nu_1 \circ \nu_2 \circ \nu_3}{f :_{\mu} A, \blacklozenge_{\nu_1}, \blacklozenge_{\nu_2}, \blacklozenge_{\nu_3} \vdash f \ : \ A \ @ \ E}$$

Modality composition $\mu \circ \nu$ is defined naturally as

$$\mu \circ [E] = [E] \qquad [E] \circ \langle D \rangle = [D, E] \qquad \langle D_1 \rangle \circ \langle D_2 \rangle = \langle D_2, D_1 \rangle$$

Effect Theories

Different effect systems collect effects as different structures such as simple rows, scoped rows, sets, multisets

This is orthogonal to their effect tracking mechanisms

[8] Morris and McKinna, “Abstracting Extensible Data Types: Or, Rows by Any Other Name”, 2019.
[9] Yoshioka, Sekiyama, and Igarashi, “Abstracting Effect Systems for Algebraic Effect Handlers”, 2024.

Effect Theories

Different effect systems collect effects as different structures such as simple rows, scoped rows, sets, multisets

This is orthogonal to their effect tracking mechanisms

$\textsc{Met}(\mathcal{X})$ is parameterised by an effect theory $\mathcal{X}$ which defines the structure of effect collections, following Morris and McKinna[8] and Yoshioka et al.[9]

[8] Morris and McKinna, "Abstracting Extensible Data Types: Or, Rows by Any Other Name", 2019.
[9] Yoshioka, Sekiyama, and Igarashi, "Abstracting Effect Systems for Algebraic Effect Handlers", 2024.

Effect Theories

Different effect systems collect effects as different structures such as simple rows, scoped rows, sets, multisets

This is orthogonal to their effect tracking mechanisms

$\textsc{Met}(\mathcal{X})$ is parameterised by an effect theory $\mathcal{X}$ which defines the structure of effect collections, following Morris and McKinna[8] and Yoshioka et al.[9]

For instance, $\mathcal{R}_{\text{scp}}$ for scoped rows and $\mathcal{S}$ for sets

We use $\textsc{Met}(\mathcal{R}_{\text{scp}})$ to encode System F^{ϵ} and $\textsc{Met}(\mathcal{S})$ to encode System C

[8] Morris and McKinna, "Abstracting Extensible Data Types: Or, Rows by Any Other Name", 2019.
[9] Yoshioka, Sekiyama, and Igarashi, "Abstracting Effect Systems for Algebraic Effect Handlers", 2024.

Rows as Modal Effects

Encoding System F^{ϵ} into $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$

Recall that effect annotations on function arrows $A \to^{E} B$ fully specify the effects

Absolute modalities also fully specify the effects

Encoding System F^{ϵ} into $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$

Recall that effect annotations on function arrows $A \to^{E} B$ fully specify the effects

Absolute modalities also fully specify the effects

$$[\![A \to^{E} B]\!] = [[\![E]\!]]([\![A]\!] \to [\![B]\!])$$

Encoding System F^{ϵ} into $\textsc{Met}(\mathcal{R}_{\text{scp}})$

Recall that effect annotations on function arrows $A \to^{E} B$ fully specify the effects

Absolute modalities also fully specify the effects

$$[\![A \to^{E} B]\!] = [[\![E]\!]]([\![A]\!] \to [\![B]\!])$$

Examples

- $$\begin{aligned} [\![\texttt{Int} \to^{\texttt{yield}} \texttt{1}]\!] &= [\texttt{yield}](\texttt{Int} \to \texttt{1}) \\ [\![\lambda^{\texttt{yield}} x^{\texttt{Int}}.\texttt{do yield } x]\!] &= \texttt{mod}_{[\texttt{yield}]}\ (\lambda x^{\texttt{Int}}.\texttt{do yield } x) \end{aligned}$$
- $$\begin{aligned} [\![\forall \varepsilon.(\texttt{Int} \to^{\varepsilon} \texttt{1}) \to \texttt{Int} \to^{\varepsilon} \texttt{1}]\!] &= \forall \varepsilon.[]([\varepsilon](\texttt{Int} \to \texttt{1}) \to [\varepsilon](\texttt{Int} \to \texttt{1})) \\ [\![\Lambda \varepsilon.\lambda f^{\texttt{Int} \to^{\varepsilon} \texttt{1}}.\lambda^{\varepsilon} x^{\texttt{Int}}.f\ x]\!] &= \Lambda \varepsilon.\texttt{mod}_{[]}\ (\lambda f^{[\varepsilon](\texttt{Int} \to \texttt{1})}.\texttt{mod}_{[\varepsilon]}\ (\lambda x^{\texttt{Int}}. \\ & \quad \texttt{let mod}_{[\varepsilon]}\ f' = f \texttt{ in } f'\ x)) \end{aligned}$$

Capabilities as Modal Effects

Encoding System C into MET($\mathcal{S}$)

$$
\begin{array}{lll}
app_C \doteq & \{(x : \texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow f(x)\} & : (\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1} \\
app'_C \doteq & \{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \text{box } \{(x : \texttt{Int}) \Rightarrow f(x)\}\} & : (f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1} \text{ at } \{f\}) \\
y :^* \texttt{Int} \Rightarrow \texttt{1} \vdash & \{(x : \texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow y(x)\} & : (\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1}
\end{array}
$$

A block construction in System C does several things:

(1) bind a both term- and type-level capability f

(2) f may be invoked in the block body

(3) any capability from the context may also be invoked in the block body

Encoding Block Constructions

A block construction in System C does several things:

(1) bind a both term- and type-level capability f
(2) f may be invoked in the block body
(3) any capability from the context may also be invoked in the block body

Encoding Block Constructions

A block construction in System C does several things:

(1) bind a both term- and type-level capability f
(2) f may be invoked in the block body
(3) any capability from the context may also be invoked in the block body

For (1), we introduce a type-level variable f^*

For (2) and (3), we use the relative modality $\langle f^* \rangle$

$$[\![(\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1}]\!] \;=\; \forall f^*.\, \langle f^* \rangle (\texttt{Int} \to [f^*](\texttt{Int} \to \texttt{1}) \to \texttt{1})$$

Encoding Block Constructions

A block construction in System C does several things:

(1) bind a both term- and type-level capability f
(2) f may be invoked in the block body
(3) any capability from the context may also be invoked in the block body

For (1), we introduce a type-level variable f^*

For (2) and (3), we use the relative modality $\langle f^* \rangle$

$$[\![(\mathtt{Int}, f : \mathtt{Int} \Rightarrow \mathtt{1}) \Rightarrow \mathtt{1}]\!] \;=\; \forall f^*.\, \langle f^* \rangle (\mathtt{Int} \to [f^*](\mathtt{Int} \to \mathtt{1}) \to \mathtt{1})$$

$$\begin{aligned} & [\![\{(x : \mathtt{Int}, f : \mathtt{Int} \Rightarrow \mathtt{1}) \Rightarrow f(x)\}]\!] \\ = \;& \Lambda f^*.\, \mathsf{mod}_{\langle f^* \rangle}\, (\lambda x^{\mathtt{Int}}.\lambda f^{[f^*](\mathtt{Int} \to \mathtt{1})}.\mathsf{let}\ \mathsf{mod}_{[f^*]}\ \hat{f} = f\ \mathsf{in}\ \hat{f}\ x) \end{aligned}$$

Encoding Boxes

Boxes in System C are encoded as absolute modalities

$$[\![(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1} \text{ at } \{f\})]\!] = \forall f^*.\, \langle f^* \rangle([f^*](\texttt{Int} \to \texttt{1}) \to [f^*](\texttt{Int} \to \texttt{1}))$$

Encoding Boxes

Boxes in System C are encoded as absolute modalities

$$[\![(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1} \text{ at } \{f\})]\!] = \forall f^*.\, \langle f^* \rangle([f^*](\texttt{Int} \to \texttt{1}) \to [f^*](\texttt{Int} \to \texttt{1}))$$

$$\begin{aligned} & [\![\{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \text{box } \{(x : \texttt{Int}) \Rightarrow f(x)\}\}]\!] \\ = \; & \Lambda f^*.\, \text{mod}_{\langle f^* \rangle}\, (\lambda f^{[f^*](\texttt{Int} \to \texttt{1})}.\text{let mod}_{[f^*]}\, \hat{f} = f \text{ in mod}_{[f^*]}\, (\lambda x.\hat{f}\, x)) \end{aligned}$$

Comparing Rows and Capabilities

By encoding both System F^{ϵ} and System C into $\textsc{Met}(\mathcal{X})$, we can easily compare them

System F^{ϵ} to $\textsc{Met}(\mathcal{R}_{\mathrm{scp}})$:

$$[\![A \to^{E} B]\!] = [[\![E]\!]]([\![A]\!] \to [\![B]\!])$$
$$[\![\forall \varepsilon.\ A]\!] = \forall \varepsilon.\ [\![A]\!]$$

System C to $\textsc{Met}(\mathcal{S})$:

$$[\![(\overline{A}, \overline{f : T}) \Rightarrow B]\!] = \forall \overline{f^*}.\ \langle \overline{f^*} \rangle (\overline{[\![A]\!]} \to \overline{[f^*][\![T]\!]} \to [\![B]\!])$$
$$[\![T \text{ at } C]\!] = [[\![C]\!]][\![T]\!]$$

Two main observations:

- different top-level modalities
- effect variables

If I Still Have Time ...

Handlers and Their Encodings

An effect handler for `yield`

$$\textsf{handle}\ (\textsf{do}\ \texttt{yield}\ 42; \textsf{do}\ \texttt{yield}\ 37; 0)\ \textsf{with}\ \{\texttt{yield}\ p\ r \mapsto p + r\ ()\}$$

Handlers and Their Encodings

An effect handler for `yield`

$$\textsf{handle}\ (\textsf{do}\ \texttt{yield}\ 42; \textsf{do}\ \texttt{yield}\ 37; 0)\ \textsf{with}\ \{\texttt{yield}\ p\ r \mapsto p + r\ ()\}$$

Encoding effect handlers in System F^{ϵ} is straightforward

Encoding effect handlers in System C is more interesting as capability-based effect systems are typically designed for lexically-scoped / named handlers

$$\textsf{try}\ \{y^{\mathtt{Int} \Rightarrow \mathtt{1}} \Rightarrow y(42); y(37); 0\}\ \textsf{with}\ \{p\ r \mapsto p + r(())\}$$

The handler introduces the capability y for invoking the operation

Encoding Handlers of System C in $\textsc{Met}(\mathcal{S})$

$$\textsf{try } \{y^{\texttt{Int}\Rightarrow\texttt{1}} \Rightarrow y(42); y(37); 0\} \textsf{ with } \{p\ r \mapsto p + r(())\}$$

There is a semantics gap between Plotkin and Pretnar's handlers and named handlers.

Instead of introducing named handlers to $\textsc{Met}(\mathcal{X})$, we use a minimal extension of local labels, inspired by Vilhena and Pottier[10] and Biernacki et al.[11]

[10] Vilhena and Pottier, "A Type System for Effect Handlers and Dynamic Labels", 2023.
[11] Biernacki, Piróg, Polesiuk, and Sieczkowski, "Abstracting algebraic effects", 2019.

Encoding Handlers of System C in $\textsc{Met}(\mathcal{S})$

$$\textsf{try } \{y^{\texttt{Int}\Rightarrow\texttt{1}} \Rightarrow y(42); y(37); 0\} \textsf{ with } \{p\ r \mapsto p + r(())\}$$

There is a semantics gap between Plotkin and Pretnar's handlers and named handlers.

Instead of introducing named handlers to $\textsc{Met}(\mathcal{X})$, we use a minimal extension of local labels, inspired by Vilhena and Pottier[10] and Biernacki et al.[11]

$$\begin{aligned}&\textsf{local } \ell_y : \texttt{Int} \Rightarrow \texttt{1} \textsf{ in handle}^{[\![C]\!]}\\ &\quad (\lambda y^{[\ell_y](\texttt{Int}\to\texttt{1})}.\textsf{let mod}_{[\ell_y]}\ \hat{y} = y \textsf{ in } \hat{y}\ 42; \hat{y}\ 37; 0)\ (\textsf{mod}_{[\ell_y]}\ (\lambda x^{\texttt{Int}}.\textsf{do}\ \ell_y\ x))\\ &\quad \textsf{with } H' : \texttt{Int} \mathbin{@} [\![C]\!]\end{aligned}$$

We simulate the capability via the function $(\textsf{mod}_{[\ell_y]}\ (\lambda x^{\texttt{Int}}.\textsf{do}\ \ell_y\ x))$

[10] Vilhena and Pottier, "A Type System for Effect Handlers and Dynamic Labels", 2023.

[11] Biernacki, Piróg, Polesiuk, and Sieczkowski, "Abstracting algebraic effects", 2019.

Encoding a Fragment of System F^{ϵ} in Monomorphic MET($\mathcal{R}_{scp}$)

One of the main selling point of modal effect types: modular effectful programming without effect variables

$$
\begin{aligned}
[\![1]\!]_E &= \langle\rangle 1 \\
[\![A \to^F B]\!]_E &= \langle E - F | F - E \rangle ([\![A]\!]_F \to [\![B]\!]_F) \\
[\![\forall . A]\!]_E &= [][\![A]\!].
\end{aligned}
$$

Come to my talk at OOPSLA, Sat 18 Oct, 10:45 - 11:00 to see more!

Encoding Variants of Koka and Effekt

We have also encoded

- System Ξ, an early core calculus of Effekt[12]
- System $F^{\epsilon+sn}$, an extension of System F^{ϵ} with named handlers and first-class names[13]

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

We proved type and semantics preservation for all the encodings

[12] Brachthäuser, Schuster, and Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism", 2020.
[13] Xie, Cong, Ikemori, and Leijen, "First-class names for effect handlers", 2022.

Insights for Language Designers

- Our encodings together demonstrate that modal effect types are as expressive as row-based and capability-based effect systems we consider.
- The encodings of System C, System Ξ, and System $F^{\epsilon+sn}$ demonstrate that we can use local labels to simulate the relatively heavyweight feature of named handlers in Effekt and Koka.
- The encoding of System $F^{\epsilon+sn}$ demonstrates that first-class handler names of Koka do not provide extra expressiveness compared to second-class local labels.
- The encoding of System C shows that instead of having a built-in form of capabilities which can appear at both term and type levels as in Effekt, we can simulate it by introducing an effect variable for each argument and wrap the argument into an absolute modality with the corresponding effect variable.
- …

Summary

More in the Papers

Modal Effect Types. OOPSLA 2025. Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen.

- Focus on ergonomics: how modal effect types enable the reuse of higher-order functions with different effects without parametric effect polymorphism
- ***Talk at OOPSLA, Sat 18 Oct, 10:45 - 11:00***

Rows and Capabilities as Modal Effects. Conditionally accepted by POPL 2026. Wenhao Tang and Sam Lindley.

- Focus on expressiveness: how modal effect types provide a unified framework for encoding and comparing different effect systems

Ongoing and Future Work

- Better type inference for modal types (as well as first-class polymorphism)
  Talk at ML family workshop, Thu 16 Oct, 13:45 - 14:15
- Higher-order effects
- Fitch-style modality elimination
- Denotational semantics and logical relations
- Other directions of encodings
- Applying the idea of relative modalities to other areas
- ...

Takeaways

Decouple effect tracking from standard type and term constructs

This decoupling provides the flexibility to simulate how effect tracking works in different effect systems

System F^{ϵ} to $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$: $\quad [\![A \to^{E} B]\!] = [[\![E]\!]]([\![A]\!] \to [\![B]\!])$

System C to $\textsc{Met}(\mathcal{S})$: $\quad [\![(\overline{A}, \overline{f : T}) \Rightarrow B]\!] = \forall \overline{f^*}.\langle \overline{f^*} \rangle (\overline{[\![A]\!]} \to \overline{[f^*][\![T]\!]} \to [\![B]\!])$