Rows and Capabilities as Modal Effects

WENHAO TANG, The University of Edinburgh, UK
SAM LINDLEY, The University of Edinburgh, UK

Effect handlers allow programmers to model and compose computational effects modularly. Effect systems statically guarantee that all effects are handled. Several recent practical effect systems are based on either row polymorphism or capabilities. However, there remains a gap in understanding the precise relationship between effect systems with such disparate foundations. The main difficulty is that in both row-based and capability-based systems, effect tracking is typically *entangled* with other features such as functions.

We propose a uniform framework for encoding, analysing, and comparing effect systems. Our framework exploits and generalises modal effect types, a recent novel effect system which *decouples* effect tracking from functions via modalities. Modalities offer fine-grained control over when and how effects are tracked, enabling us to express different strategies for effect tracking. We give encodings as macro translations from existing row-based and capability-based effect systems into our framework and show that these encodings preserve types and semantics. Our encodings reveal the essence of effect tracking mechanisms in different effect systems, enable a direct analysis on their differences, and provide practical insights on language design.

CCS Concepts: • **Theory of computation** → **Type structures**; **Type theory**; **Control primitives**.

Additional Key Words and Phrases: effect handlers, effect types, modal types

1 Introduction

Effect handlers [34] provide a powerful abstraction to define and compose computational effects including state, concurrency, and probability. Effect systems statically ensure that all effects used in a program are handled. The literature includes much work on effect systems for effect handlers based on a range of different theoretical foundations. Two of the most popular and well-studied approaches are row-based effect systems [16, 24, 28] and capability-based effect systems [5–7].

Row-based effect systems, as in the languages Koka [24, 41], Links [16], and Frank [28], follow the traditional monadic reading of effects: effects are what computations do when they run. They treat effect types as a row of effects and annotate each function arrow with an effect row. For modularity, they implement *parametric effect polymorphism* via row polymorphism [23, 36]. For example, a standard application function in System F^{ϵ} [40], a core calculus of Koka, has type:

$$\forall \varepsilon.(\mathtt{Int} \to^{\varepsilon} \mathtt{1}) \to \mathtt{Int} \to^{\varepsilon} \mathtt{1}$$

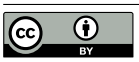
It is polymorphic in its effects ε, which must agree with the effect performed by its first argument.

Capability-based effect systems, as in the language Effekt [6, 7] and an extension to Scala 3 [5], adopt a contextual reading of effects: effects are capabilities provided by the context. Treating effects as capabilities enables a notion of *contextual effect polymorphism* [7] which allows effect-polymorphic reuse of functions without effect variables. For example, an uncurried application function in System C [6], a core calculus of Effekt, has type:

$$(f : \mathtt{Int} \Rightarrow \mathtt{1}, \mathtt{Int}) \Rightarrow \mathtt{1}$$

The argument f is a capability. It is a second-class function that cannot be returned as a value. It can use any capabilities the context provides. We write $\Rightarrow$ for second-class functions. For a curried

Authors' Contact Information: Wenhao Tang, wenhao.tang@ed.ac.uk, The University of Edinburgh, UK; Sam Lindley, sam.lindley@ed.ac.uk, The University of Edinburgh, UK.



application function, which requires returning a function, we must capture capabilities in types:

$$(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1}\ \textbf{at}\ \{f\})$$

Its result has type $(\texttt{Int} \Rightarrow \texttt{1}\ \textbf{at}\ \{f\})$. As well as specifying an argument and result types as usual, this type also includes a *capture set* $\{f\}$ which records that the returned function may use the capability f bound by the argument type $(f : \texttt{Int} \Rightarrow \texttt{1})$.

Though row-based and capability-based effect systems are both well-studied, their relationship is not. In this paper, we aim to bridge this gap in the literature by encoding both styles of effect systems into a uniform framework. Yoshioka et al. [43] propose a parameterised calculus which can be instantiated to various row-based effect systems, but they point out that it is challenging future work to extend their approach to capability-based effect systems. Row-based and capability-based effect systems differ significantly in both theoretical foundations and interpretations of effects. Moreover, their mechanisms for tracking effects are *entangled* with other features such as functions. For instance, as we have seen above, a function arrow in System F^ϵ is not only a standard function type but also provides effect annotations. Similarly, a function arrow in System C may bind capabilities. The entanglement of effect tracking with such features is the central challenge in analysing the differences between such effect systems.

An alternative foundation for effect systems has recently emerged in the form of *modal effect types* (Met) [38], a novel approach to effect systems based on multimodal type theory [14, 15, 21]. Met *decouples* effect tracking from standard type and term constructs via modalities. For instance, an application function in Met has a plain function type $(\texttt{Int} \to \texttt{1}) \to \texttt{Int} \to \texttt{1}$. This type imposes no restriction on how effects from the context may be used. To control the use of effects, we can add modalities to the type. For example, the type $[\texttt{yield}](\texttt{Int} \to \texttt{1}) \to \texttt{Int} \to \texttt{1}$ restricts the argument function to use only the operation `yield` by wrapping it with the *absolute modality* $[\texttt{yield}]$ (modalities have higher precedence than function arrows); the type $[](\texttt{Int} \to \texttt{1}) \to [](\texttt{Int} \to \texttt{1})$ restricts both the argument and result functions to be pure.

Tang et al. [38] focus on the pragmatics of Met, especially how modalities enable concise type signatures of higher-order functions without losing modularity. In this paper we exploit the observation that the decoupling of effect tracking via modalities leads to a tangible increase in flexibility and expressivity compared to typical effect systems whose effect tracking is entangled with other features. Such decoupling allows us to encode a range of effect systems, including those based on rows and capabilities, in a uniform framework.

We introduce Met($\mathcal{X}$), a System F-style core calculus with modal effect types parameterised by an *effect structure* $\mathcal{X}$. The effect structure is our main extension to Met [38]. An effect structure, inspired by prior work on abstracting row and effect types [17, 30, 43], defines the structure of effect collections. Met hardwires the underlying effect structure to scoped rows [23]. In contrast, Met($\mathcal{X}$) allows us to smoothly account for the different treatments of effect collections adopted by different effect systems, such as sets [1, 6], simple rows [30], and scoped rows [23, 24, 28]. Parameterising by the effect structure enables us to separate the bureaucracy of managing effect collections from our main concern which is how to use modalities to encode different effect tracking mechanisms.

Met($\mathcal{X}$) has two further extensions to Met. The first extension is *modality-parameterised handlers*. This is a natural generalisation of effect handlers to be parameterised by a modality which is used to wrap continuations. This extension is crucial for the encodings of System F^ϵ and System C as we will see in Section 2.5. The second extension is *local labels*, a minimal extension which allows us to dynamically generate operation labels [11]. This extension is crucial for encoding named handlers [4, 7, 44] (also called lexically-scoped handlers) as adopted in some languages, especially those with capability-based effect systems like Effekt.

As the main novelty of this paper, we encode, as *macro translations* [12], various effect systems based on rows and capabilities into our uniform framework $\textsc{Met}(\mathcal{X})$. We prove that our encodings preserve typing and operational semantics. Our encodings do not heavily alter the structure of programs but mostly merely insert terms for manipulating modalities; our semantics preservation theorems establish a strong correspondence between the behaviours of source calculi and their translations. Our primary case studies are encodings of System F^{ϵ} [40], a core calculus of Koka with a row-based effect system, and of System C [6], a core calculus of Effekt with a capability-based effect system. By encoding effect systems into a uniform framework, we can directly reason about the differences the effect tracking mechanisms of different effect systems (Sections 2.4 and 2.5.4). Our encodings also offer practical insights for language designers (Section 6.3).

Beyond analysing differences between effect systems, $\textsc{Met}(\mathcal{X})$ opens up interesting future research directions. First, $\textsc{Met}(\mathcal{X})$ gives a uniform intermediate representation for different effect systems which enables us to design type-directed optimisations without restricting ourselves to a specific effect system. Second, $\textsc{Met}(\mathcal{X})$ allows us to design a new effect system by directly giving its encoding into $\textsc{Met}(\mathcal{X})$ instead of starting from scratch. Type soundness and effect safety of $\textsc{Met}(\mathcal{X})$ guarantee the corresponding properties hold for the new effect system.

The main contributions of this paper are as follows.

- We give a high-level overview of $\textsc{Met}(\mathcal{X})$ and a high-level overview of how to encode row-based and capability-based effect systems into $\textsc{Met}(\mathcal{X})$ which we use to compare row-based and capability-based effect systems (Section 2).
- We formally define $\textsc{Met}(\mathcal{X})$ (Section 3) including our three extensions to Met: effect structures, modality-parameterised handlers, and local labels. We prove type soundness and effect safety of $\textsc{Met}(\mathcal{X})$ for any effect structure $\mathcal{X}$ satisfying certain natural validity conditions.
- We formally define the encoding of System F^{ϵ}, a core calculus with a row-based effect system à la Koka, into $\textsc{Met}(\mathcal{R}_{\text{scp}})$ with the theory $\mathcal{R}_{\text{scp}}$ for scoped rows (Section 4). We prove the encoding preserves types and semantics.
- We formally define the encoding of System C, a core calculus with a capability-based effect system à la Effekt, into $\textsc{Met}(\mathcal{S})$ with the theory $\mathcal{S}$ for sets (Section 5). We prove the encoding preserves types and semantics.
- We discuss encodings of further effect systems, practical insights for language design provided by our encodings, as well as potential extensions to $\textsc{Met}(\mathcal{X})$ (Section 6).

Section 7 discusses related and future work. The full specifications, proofs, and appendices can be found in the extended version of the paper [37].

2 Overview

In this section we give a high-level overview of the main ideas of the paper. We begin with a brief introduction to modal effects [38] in $\textsc{Met}(\mathcal{X})$ and examples of effect theories $\mathcal{X}$. We briefly describe the row-based effect system of System F^{ϵ} [40] and the capability-based effect system of System C [6] along with their encodings into $\textsc{Met}(\mathcal{X})$. We use these encodings to directly compare the different systems in a uniform framework. We specifically consider encodings of the different kinds of effect handlers provided by the different systems. We also briefly discuss the results of encoding other effect systems in $\textsc{Met}(\mathcal{X})$.

2.1 Modal Effects and Met($\mathcal{X}$)

Met($\mathcal{X}$) is a System F-style core calculus. Every well-typed term in System F is also well-typed in Met($\mathcal{X}$). For example, we may define a higher-order application function as follows.

$$app_{\text{Met}(\mathcal{X})} \doteq \lambda f^{\texttt{Int}\to 1}.\lambda x^{\texttt{Int}}.f\ x \ :\ (\texttt{Int} \to 1) \to \texttt{Int} \to 1$$

We use meta-level macros defined by $\doteq$ in red to refer to code snippets.

2.1.1 Effect Contexts. Met($\mathcal{X}$) adopts a contextual reading of effects. Effectful operations are ascribed a type signature, either globally or locally. For our examples we begin by assuming global operations $\texttt{yield} : \texttt{Int} \twoheadrightarrow 1$ and $\texttt{ask} : 1 \twoheadrightarrow \texttt{Int}$. Typing judgements include an *ambient effect context* which tracks the operations that may be performed. Consider the following function.

$$\vdash gen_{\text{Met}(\mathcal{X})} \doteq \lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x \ :\ \texttt{Int} \to 1 \ @\ \texttt{yield}$$

It has type $\texttt{Int} \to 1$. When applied it performs the `yield` operation using the **do** syntax. The judgement specifies the effect context with the syntax $@\,\texttt{yield}$, which tracks the possibility of performing `yield`. We can now apply $app_{\text{Met}(\mathcal{X})}$ to $gen_{\text{Met}(\mathcal{X})}$ and 42 as follows.

$$\vdash (\lambda f^{\texttt{Int}\to 1}.\lambda x^{\texttt{Int}}.f\ x)\ (\lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x)\ 42 \ :\ 1 \ @\ \texttt{yield}$$

There is a natural notion of subeffecting on effect contexts. The following judgement is also valid.

$$\vdash \lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x \ :\ \texttt{Int} \to 1 \ @\ \texttt{yield}, \texttt{ask}$$

2.1.2 Absolute Modalities. Effect contexts specified by $@\,E$ belong to typing judgements instead of types. As discussed in Section 1, Met($\mathcal{X}$) uses modalities to track effects in types. An *absolute modality* $[E]$ allows us to specify a new effect context E in types different from the ambient one. For example, consider the following typing derivation.

$$\frac{\lock_{[\texttt{yield}]} \vdash \lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x \ :\ \texttt{Int} \to 1 \ @\ \texttt{yield}}{\vdash gen'_{\text{Met}(\mathcal{X})} \doteq \mathbf{mod}_{[\texttt{yield}]}\ (\lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x) \ :\ [\texttt{yield}](\texttt{Int} \to 1) \ @\ F}$$

This term has type $[\texttt{yield}](\texttt{Int} \to 1)$. We highlight modalities in blue when they appear in types. The syntax $\mathbf{mod}_{[\texttt{yield}]}$ introduces an absolute modality $[\texttt{yield}]$ which specifies a singleton effect context of `yield` and uses it to override the ambient effect context F. The typing judgement of the premise uses the new effect context `yield` as its ambient effect context. The lock $\lock_{[\texttt{yield}]}$ tracks the switch of the effect context and controls the accessibility of variables on the left of it. Only variables that are known not to use effects other than `yield` may be used. This is important to ensure effect safety. For example, consider the following invalid judgement.

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

This program is unsafe as f may invoke `ask` which we must not use under effect context `yield` specified by the modality $[\texttt{yield}]$. Met($\mathcal{X}$) rejects this judgement as it relies on the following invalid judgement for the inner function.

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

This typing judgement is invalid as the lock $\lock_{[\texttt{yield}]}$ prevents the use of f. To make it valid, we can annotate the binding of f with the modality $[\texttt{yield}]$ as $f :_{[\texttt{yield}]} \texttt{Int} \to 1$. This annotation tracks that the function f may only use the operation `yield`. Such annotated bindings are introduced by modality elimination. For instance, we can eliminate the modality of $gen'_{\text{Met}(\mathcal{X})}$ and then apply it via the **let mod** syntax as follows (where we elide the typing of the bound term).

$$\frac{\ldots \qquad f :_{[\texttt{yield}]} \texttt{Int} \to 1 \vdash f\ 42 \ :\ 1 \ @\ \texttt{yield}}{\vdash \mathbf{let\ mod}_{[\texttt{yield}]}\ f = \mathbf{mod}_{[\texttt{yield}]}\ (\lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x)\ \mathbf{in}\ f\ 42 \ :\ 1 \ @\ \texttt{yield}}$$

The term $\lambda x^{\mathtt{Int}}.\mathbf{do}\ \mathtt{yield}\ x$ inside the modality $[\mathtt{yield}]$ is bound to f. The binding of f is annotated with this absolute modality. Consequently, the use of f in $f\ 42$ requires the ambient effect context to contain the operation `yield`. In general, whether a variable binding $f :_{\mu} A$ can be used after a lock 🔒_{ν} is controlled by a modality transformation relation which we will introduce in Section 3.3.

2.1.3 Relative Modalities. As well as being able to specify a fresh effect context from scratch with an absolute modality, $\textsc{Met}(\mathcal{X})$ also has *relative modalities* $\langle D \rangle$ which allow us to extend the ambient effect context with an *extension* D. For instance, consider the following derivation.

$$\frac{\text{🔒}_{\langle \mathtt{yield} \rangle} \vdash \lambda x^{\mathtt{Int}}.\mathbf{do}\ \mathtt{yield}\ (\mathbf{do}\ \mathtt{ask}\ ()) \ :\ \mathtt{Int} \to \mathtt{1}\ @\ \mathtt{yield}, \mathtt{ask}}{\vdash \mathbf{mod}_{\langle \mathtt{yield} \rangle}\ (\lambda x^{\mathtt{Int}}.\mathbf{do}\ \mathtt{yield}\ (\mathbf{do}\ \mathtt{ask}\ ())) \ :\ \langle \mathtt{yield} \rangle (\mathtt{Int} \to \mathtt{1})\ @\ \mathtt{ask}}$$

The relative modality $\langle \mathtt{yield} \rangle$ extends the ambient effect context `ask` with the operation `yield`. Consequently, the inside function can use both operations. Relative modalities are especially useful for giving composable types to effect handlers. We refer to Tang et al. [38] for further details. We use relative modalities in the encoding of System C as we will see in Section 2.3.

2.1.4 Effect Structures. Improving on $\textsc{Met}$, we parameterise $\textsc{Met}(\mathcal{X})$ by an effect structure $\mathcal{X}$ which defines the well-formedness relations and equivalence relations for extensions and effect contexts as well as a subeffecting relation $E \leqslant F$. In the remainder of the overview, we will use two effect structures: $\mathcal{S}$, which models effect collections as sets of operations, to encode capability sets in System C, and $\mathcal{R}_{\text{scp}}$, which models effect collections as scoped rows of operations, to encode effect rows in System F^{ϵ}. Sets are unordered and allow only one occurrence of each label, whereas scoped rows allow repeated labels and identify rows up to reordering of non-identical labels. Both theories support effect variables. Theory $\mathcal{S}$ allows arbitrary numbers of effect variables while theory $\mathcal{R}_{\text{scp}}$ only allows at most one effect variable in each row following row polymorphism [23, 36].

2.2 Rows as Modal Effects

Koka [25] has an effect system based on scoped rows [23]. System F^{ϵ} [40] is a core calculus underlying Koka. To encode System F^{ϵ}, we use the effect structure $\mathcal{R}_{\text{scp}}$ of scoped rows.

Function types in System F^{ϵ} have the form $A \to^{E} B$, where E is an effect row that specifies the effects that the function may use. Effect types in System F^{ϵ} are entangled with function types. The key idea of our encoding is to decouple the effect type E from the function arrow via an absolute modality in $\textsc{Met}(\mathcal{R}_{\text{scp}})$. Writing $[\![-]\!]$ for translations, we translate a function type as follows.

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

An effectful function in System F^{ϵ} is decomposed into an absolute modality and a standard function in $\textsc{Met}(\mathcal{R}_{\text{scp}})$. For instance, consider the following first-order effectful function in System F^{ϵ} which invokes the operation `yield` from Section 2.1.

$$gen_{\mathsf{F}^{\epsilon}} \doteq \lambda^{\mathtt{yield}} x^{\mathtt{Int}}.\mathbf{do}\ \mathtt{yield}\ x \ :\ \mathtt{Int} \to^{\mathtt{yield}} \mathtt{1}$$

(Each λ-abstraction in System F^{ϵ} is annotated with an effect row.) The translation of $gen_{\mathsf{F}^{\epsilon}}$ is exactly the function $gen'_{\textsc{Met}(\mathcal{X})}$ defined in Section 2.1.2. We repeat its definition here for easy reference.

$$[\![gen_{\mathsf{F}^{\epsilon}}]\!] = \mathbf{mod}_{[\mathtt{yield}]}\ (\lambda x^{\mathtt{Int}}.\mathbf{do}\ \mathtt{yield}\ x) \ :\ [\mathtt{yield}](\mathtt{Int} \to \mathtt{1})$$

On the term level, we insert a modality introduction $\mathbf{mod}_{[\mathtt{yield}]}$ for the λ-abstraction, corresponding to the type-level modality $[\mathtt{yield}]$. We colour **mod** in grey in the translations. The black parts remain terms with valid syntax and provide intuitions on the translation. Remember that the modality $[\mathtt{yield}]$ is a first-class type constructor and not part of the function type.

As a more non-trivial example including both higher-order functions and function application, consider the effect-polymorphic application function in System F^ϵ from Section 1.

$$app_{\mathsf{F}^\epsilon} \doteq \Lambda\varepsilon^{\mathsf{Effect}}.\lambda f^{\mathtt{Int}\to^{\varepsilon}\mathtt{1}}.\lambda x^{\mathtt{Int}}.f\ x \ :\ \forall\varepsilon.(\mathtt{Int}\to^{\varepsilon}\mathtt{1})\to\mathtt{Int}\to^{\varepsilon}\mathtt{1}$$

This function abstracts over an effect variable ε which stands for the effects performed by the argument f. Both f and the inner λ-abstraction are annotated with ε as f is invoked so the effects must match up. The outer λ-abstraction is pure as partial application is pure. The encoding of $app_{\mathsf{F}^\epsilon}$ in $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$ is as follows.

$$\begin{aligned}[\![app_{\mathsf{F}^\epsilon}]\!] \ =\ & \Lambda\varepsilon^{\mathsf{Effect}}.\mathbf{mod}_{[]}\ (\lambda f^{[\varepsilon](\mathtt{Int}\to\mathtt{1})}.\mathbf{mod}_{[\varepsilon]}\ (\lambda x^{\mathtt{Int}}.\mathbf{let}\ \mathbf{mod}_{[\varepsilon]}\ f' = f\ \mathbf{in}\ f'\ x))\\ :\ & \forall\varepsilon^{\mathsf{Effect}}.[]([\varepsilon](\mathtt{Int}\to\mathtt{1})\to[\varepsilon](\mathtt{Int}\to\mathtt{1}))\end{aligned}$$

Each function arrow is associated with an absolute modality reflecting the effects performed by that function. For the pure function arrow in the middle, we use the empty absolute modality $[]$. The type abstraction $\Lambda\varepsilon$ and quantifier $\forall\varepsilon$ are preserved. We omit kinds when obvious. In the term, in addition to modality introduction, we also insert a modality elimination for f before applying it to x. The use of f' requires that the effect variable ε is present in the effect context.

Our term translation from System F^ϵ to $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$ explicitly decouples the effect tracking mechanism of System F^ϵ from function abstraction and application. This reveals the essence of effect tracking in System F^ϵ. Each λ-abstraction $\lambda^E x.M$ in System F^ϵ is encoded in $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$ by inserting a modality introduction $\mathbf{mod}_{[\![E]\!]}$. This demonstrates that a function in System F^ϵ carries its effects. Each function application $V\ W$ in System F^ϵ is encoded by inserting a modality elimination $\mathbf{let}\ \mathbf{mod}_{[\![E]\!]}\ f = [\![V]\!]\ \mathbf{in}\ f\ [\![W]\!]$ for function V of type $A\to^E B$. This demonstrates that when a function is invoked in System F^ϵ, we need to provide all effects it may use, as the elimination of $[[\![E]\!]]$ and use of f together require $[\![E]\!]$ to be present in the effect context.

We give the full encoding of System F^ϵ into $\textsc{Met}(\mathcal{R}_{\mathsf{scp}})$ in Section 4.

2.3 Capabilities as Modal Effects

Effekt [8] has an effect system based on capabilities. System C [6] is a core calculus underlying Effekt. Since System C tracks capabilities as sets, we use the effect structure $\mathcal{S}$ of sets to encode it.

Functions in System C are called *blocks*. Blocks are second-class in that they must be fully applied and cannot be returned. Capabilities are introduced as block variables. Unlike row-based effect systems which have a separate notion of operation labels, System C interprets effects as capabilities provided by the context. A capability can only be used if it is in scope.

2.3.1 First-Order Blocks. Let us start with a simple example. Supposing we have a capability $y : \mathtt{Int} \Rightarrow \mathtt{1}$ (for yielding integers) in the context, we can construct the following block.

$$y :^* \mathtt{Int}\Rightarrow\mathtt{1} \ \vdash\ gen_C \doteq \{(x:\mathtt{Int})\Rightarrow y(x)\}\ :\ \mathtt{Int}\Rightarrow\mathtt{1}\ \mid\ \{y\}$$

The star $*$ on the binding of y indicates that this block variable is a capability. Braces delimit blocks. Arguments are wrapped in parentheses. Double arrows emphasise that blocks are second-class. The block applies the capability y from the context to the argument x. The typing judgement tracks a capability set $\{y\}$, which contains all capabilities that the block may use. The block arrow itself has no capability annotation. The above block is simply encoded as a λ-abstraction in $\textsc{Met}(\mathcal{S})$.[1]

$$y^* : \mathsf{Effect}, y : [y^*](\mathtt{Int}\to\mathtt{1}), \hat{y} :_{[y^*]} \mathtt{Int}\to\mathtt{1} \ \vdash\ [\![gen_C]\!] \ =\ \lambda x^{\mathtt{Int}}.\hat{y}\ x \ :\ \mathtt{Int}\to\mathtt{1}\ @\ y^*$$

The most interesting aspect of the encoding is how we encode the capability $y : \mathtt{Int}\Rightarrow\mathtt{1}$ in the context. A capability y in System C can appear as both a type and a term. We introduce an effect

[1] If we strictly follow the encoding of System C in Section 5.2, there would be an extra identity modality for the translated function. This modality is crucial for keeping the encoding systematic. We omit such identity modalities in the overview.

variable y^* of kind Effect to represent it at the type level. We omit kinds in the context when obvious. We encode the capability y itself as a term variable of type $[y^*](\texttt{Int} \to \texttt{1})$, where the absolute modality makes sure that whenever y is invoked the effect variable y^* must be present in the effect context. To avoid repeatedly writing modality eliminations, the modality of y is immediately eliminated and bound to $\hat{y}$ after y is introduced. The translation of the block body directly applies $\hat{y}$ to x. The effect variable y^* must be in the effect context specified by $@\ y^*$ because $\hat{y}$ is used.

2.3.2 Boxes. In System C we can turn a second-class block into a first-class value by boxing it.

$$y :^* \texttt{Int} \Rightarrow \texttt{1} \ \vdash\ gen'_C \doteq \textbf{box}\ \{(x : \texttt{Int}) \Rightarrow y(x)\}\ :\ \texttt{Int} \Rightarrow \texttt{1}\ \textbf{at}\ \{y\}$$

This typing judgement has no capability set as it is for values which are always pure in System C. The value has type $\texttt{Int} \Rightarrow \texttt{1}$ **at** $\{y\}$, which means it is a boxed block of type $\texttt{Int} \Rightarrow \texttt{1}$ with capability set $\{y\}$. The block may only use the capability y. We can unbox a boxed block V via **unbox** V which gives back a second-class block. We simply encode boxing and unboxing as modality introduction and elimination in $\textsc{Met}(\mathcal{S})$. For instance, we encode gen'_C as follows.

$$y^*, y : [y^*](\texttt{Int} \to \texttt{1}), \hat{y} :_{[y^*]} \texttt{Int} \to \texttt{1} \ \vdash\ [\![gen'_C]\!] \ =\ \mathbf{mod}_{[y^*]}\ (\lambda x^{\texttt{Int}}.\hat{y}\ x)\ :\ [y^*](\texttt{Int} \to \texttt{1})\ @\ \cdot$$

The capability set annotation **at** $\{y\}$ in the type is encoded as the absolute modality $[y^*]$. The encoding shows the connection between boxes of System C and modalities, supporting the claim of Brachthäuser et al. [6] that boxes of System C are inspired by modal connectives [9].

2.3.3 Higher-Order Blocks. The situation become more involved when we consider higher-order blocks that take other blocks as arguments. This is because System C entangles the introduction and tracking of capabilities with blocks, especially their construction and application.

Let us consider the uncurried and curried application functions (blocks) introduced in Section 1.

$$\begin{aligned} app_C &\doteq \{(x : \texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow f(x)\} &&: (\texttt{Int}, f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1} \\ app'_C &\doteq \{(f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \textbf{box}\ \{(x : \texttt{Int}) \Rightarrow f(x)\}\} &&: (f : \texttt{Int} \Rightarrow \texttt{1}) \Rightarrow (\texttt{Int} \Rightarrow \texttt{1}\ \textbf{at}\ \{f\}) \end{aligned}$$

These are block constructions. The first block app_C binds the integer parameter x first because System C requires value parameters like x to appear before blocks parameters like f in a parameter list. In addition to behaving like standard λ-abstractions, block constructions also play an important role in capability tracking. Specifically:

(1) Both app_C and app'_C bind a capability $f : \texttt{Int} \Rightarrow \texttt{1}$ for their block bodies. This capability f can also be used in the type as shown in the type of app'_C.
(2) For soundness, System C assumes that this new capability f is called directly at least once in the block body even if f may actually not be used. (The capability f is indeed called directly in app_C but not so in $app_{C'}$ as being boxed.) Consequently, the capability f is always added to the capability set of the block body tracked by the typing judgement.
(3) In addition to the new capability f, both app_C and app'_C allow any capability from the context to be called as well.

Our encoding of block constructions in $\textsc{Met}(\mathcal{S})$ takes account of these three constraints and exposes them explicitly via modalities. For instance, app_C is encoded as follows.

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

For (1), in order to allow the term variable f to appear in types, we introduce an effect variable f^* and wrap the type $\texttt{Int} \to \texttt{1}$ of the argument f with an absolute modality $[f^*]$. The effect variable f^* represents the term variable f at the level of types. Additionally, we immediately eliminate the modality of f to $\hat{f}$. As a result, in the context of the application $\hat{f}\ x$ we have three bindings of f^*,

f, and $\hat{f}$, consistent with the translation of capability y as shown in Section 2.3.1. For (2) and (3), we use a relative modality $\langle f^* \rangle$ to wrap the whole function type. The relative modality adds the effect variable f^* to the ambient effect context for the function to use, in accordance with (2). The relative modality also still allows the function to use effects from the ambient effect context as we have seen in Section 2.1.3, in accordance with (3).

The translation of app'_C is similar.

$$\begin{array}{rcl} [\![app'_C]\!] & = & \Lambda f^*.\mathbf{mod}_{\langle f^* \rangle}\ (\lambda f^{[f^*](\mathtt{Int}\to 1)}.\mathbf{let}\ \mathbf{mod}_{[f^*]}\ \hat{f} = f\ \mathbf{in}\ \mathbf{mod}_{[f^*]}\ (\lambda x.\hat{f}\ x)) \\ & : & \forall f^*.\langle f^* \rangle([f^*](\mathtt{Int}\to 1)\to[f^*](\mathtt{Int}\to 1)) \end{array}$$

In general, the translation of block types from System C to $\textsc{Met}(\mathcal{S})$ is as follows, where we let A and B range over value types and let T range over block types.

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

A block type is decomposed into a standard function type with extra modalities and type quantifiers, which makes explicit exactly how System C introduces and tracks capabilities.

2.3.4 Block Calls. Blocks must be fully applied. Assuming we have a capability $y : \mathtt{Int} \Rightarrow \mathtt{1}$ in the context, we can apply the blocks app_C and app'_C to the block gen_C as follows.

$$\begin{array}{lll} y :^* \mathtt{Int} \Rightarrow \mathtt{1} \vdash app_C(gen_C, 42) & : \mathtt{1} & \mid \{y\} \\ y :^* \mathtt{Int} \Rightarrow \mathtt{1} \vdash app'_C(gen_C) & : \mathtt{Int} \Rightarrow \mathtt{1}\ \mathbf{at}\ \{y\} & \mid \{y\} \end{array}$$

(As blocks must be fully applied, we must additionally pass an integer to app_C — in this case 42.) These are block calls. Similar to block constructions, block calls in System C not only pass arguments to a block but also play an important role in capability tracking. Specifically:

(1) Recall that both app_C and app'_C bind a capability f. Consequently, when calling them with gen_C, System C substitutes f with the capability set $\{y\}$ of gen_C in types. This is reflected by **at** $\{y\}$ in the type of calling app'_C (before substitution it was **at** $\{f\}$).

(2) Recall that System C assumes the capability f bound by app_C and app'_C is called directly. Consequently, the capability set of the whole block call must be extended with the capability set $\{y\}$ of the argument gen_C. This is reflected by the fact that both typing judgements track the capability sets $\{y\}$ even though the application of app'_C does not call y directly.

Our encoding of block calls in $\textsc{Met}(\mathcal{S})$ takes account of these two constraints and exposes them explicitly via modalities. For instance, our example application of app_C is encoded as follows.

$$\begin{array}{l} y^*, y : [y^*](\mathtt{Int}\to\mathtt{1}), \hat{y} :_{[y^*]} \mathtt{Int}\to\mathtt{1} \vdash \\ \qquad \mathbf{let}\ \mathbf{mod}_{\langle y^* \rangle}\ f = [\![app_C]\!]\ y^*\ \mathbf{in}\ f\ 42\ (\mathbf{mod}_{[y^*]}\ [\![gen_C]\!]) \ : \mathtt{1}\ @\ y^* \end{array}$$

For (1), recall that in the translation $[\![app_C]\!]$ we bind an effect variable f^* to represent the capability f and wrap the argument type with an absolute modality $[f^*]$. Thus for the application of $[\![app_C]\!]$, we instantiate the effect variable f^* with y^* and box the argument $[\![gen_C]\!]$ with the absolute modality $[y^*]$. For (2), the elimination of the relative modality $\langle y^* \rangle$ of $[\![app_C]\!]\ y^*$ and the use of f ensure that y^* must be present in the effect context.

The translation of the call of app'_C is similar.

$$\begin{array}{l} y^*, y : [y^*](\mathtt{Int}\to\mathtt{1}), \hat{y} :_{[y^*]} \mathtt{Int}\to\mathtt{1} \vdash \\ \qquad \mathbf{let}\ \mathbf{mod}_{\langle y^* \rangle}\ f = [\![app'_C]\!]\ y^*\ \mathbf{in}\ f\ (\mathbf{mod}_{[y^*]}\ [\![gen_C]\!]) \ : [y^*](\mathtt{Int}\to\mathtt{1})\ @\ y^* \end{array}$$

As with the encoding of Section 2.2, the encoding of System C in $\textsc{Met}(\mathcal{S})$ helps elucidate exactly how the capability tracking of System C is entangled with constructs like block constructions and calls. Modality introduction and elimination reveal the hidden mechanisms.

We give the full encoding of System C into $\textsc{Met}(\mathcal{S})$ in Section 5.

2.4 Comparing Rows and Capabilities

As a uniform framework, $\textsc{Met}(\mathcal{X})$ allows us to directly compare how effect tracking differs in different effect systems without dealing with the subtleties in their typing and reduction rules.

For instance, let us compare the encoding of function types and polymorphic types in System F^ϵ with the encoding of block types and box types in System C.

$$\begin{aligned}
\text{System } \mathsf{F}^\epsilon \text{ to } \textsc{Met}(\mathcal{R}_{\text{scp}}): \qquad [\![A \to^E B]\!] &= [\![E]\!]([\![A]\!] \to [\![B]\!]) \\
[\![\forall \varepsilon. A]\!] &= \forall \varepsilon. [\![A]\!] \\
\text{System C to } \textsc{Met}(\mathcal{S}): \qquad [\![(\overline{A}, \overline{f : T}) \Rightarrow B]\!] &= \forall \overline{f^*}. \langle \overline{f^*} \rangle (\overline{[\![A]\!]} \to \overline{[f^*][\![T]\!]} \to [\![B]\!]) \\
[\![T \textbf{ at } C]\!] &= [\![C]\!][\![T]\!]
\end{aligned}$$

From the encodings we can immediately observe two key differences of System F^ϵ and System C.

(1) The encoding of function types in System F^ϵ is wrapped with an absolute modality, whereas the encoding of a block type in System C is wrapped with a relative modality. The encoding of box types in System C is wrapped with an absolute modality. The different modalities reveal a fundamental difference between the meanings of functions in System F^ϵ and blocks in System C: functions in System F^ϵ can only use those effects specified in their types, whereas blocks in System C can use arbitrary effects from the context unless they are boxed.

(2) The encoding of block types in System C binds a list of effect variables and wraps each block argument type with an absolute modality of the corresponding effect variable, whereas the encoding of a function type in System F^ϵ is much less involved. Only the encoding of polymorphic types in System F^ϵ binds effect variables. The difference in the treatment of argument types reveals that capabilities in System C act as an implicit form of parametric polymorphism, abstracting the capabilities used by each block variable. This explains why capability-based effect systems do not require explicit effect variables in many cases where row-based effect systems do.

2.5 Encoding Effect Handlers

We have seen how effectful functions in System F^ϵ and System C are encoded in $\textsc{Met}(\mathcal{X})$. These are the most important parts of our encodings, as most effect systems track effects by giving different intepretations to functions. Though all effect systems discussed in this paper support effect handlers, the same ideas apply equally to traditional effect systems for languages with only built-in effects. Nonetheless, the encodings of effect handlers in System F^ϵ and System C are interesting and reveal fundamental differences between the typing and semantics of effect handlers in these two calculi. In this section, we first briefly review what effect handlers are and then show how effect handlers in System F^ϵ and System C are encoded.

2.5.1 Effect Handlers in $\textsc{Met}(\mathcal{X})$. Effect handlers allow us to customise how to handle effectful operations. For instance, we can write a handler to handle the `yield` operation defined in Section 2.1 by summing up all yielded integers as follows.

$$sum_{\textsc{Met}(\mathcal{X})} \doteq \textbf{handle } (\textbf{do } \texttt{yield } 42; \textbf{do } \texttt{yield } 37; 0) \textbf{ with } \{\texttt{yield } p\ r \mapsto p + r\ ()\}$$

The computation **do** `yield` 42; **do** `yield` 37; 0 is handled by the handler $\{\texttt{yield } p\ r \mapsto p + r\ ()\}$. The handler consists of one operation clause for the operation `yield`. In this operation clause, the variable p of type `Int` is bound to the parameter of the `yield` operation, and the variable r of type $\texttt{1} \to \texttt{Int}$ is bound to its recursively-handled continuation. (This kind of recursive handling is known as *deep handlers* [20] in the literature.) For instance, when the first `yield` operation is handled, p is 42 and r is the continuation $\lambda y^{\texttt{1}}.\textbf{handle } (\textbf{do } \texttt{yield } 37; 0) \textbf{ with } \{\texttt{yield } p\ r \mapsto p + r\ ()\}$. The handler clause adds the yielded integer p to the result of the continuation r, thus returning the

sum of all handled operations. The above program reduces to 79. Effect handlers also have a return clause which we omit here, but describe in Section 3.

2.5.2 Encoding Effect Handlers in System F^ϵ. System F^ϵ does not use the **handle with** syntax. Instead, a handler in System F^ϵ is defined as a handler value, which is a function that takes an argument to handle. Consider the following polymorphic handler for the `yield` operation.

$$sum_{F^\epsilon} \doteq \Lambda\varepsilon.\mathbf{handler}\ \{\mathtt{yield}\ p\ r \mapsto p + r\ ()\} \quad : \quad \forall\varepsilon.(\mathtt{1} \to^{\mathtt{yield},\varepsilon} \mathtt{Int}) \to^{\varepsilon} \mathtt{Int}$$

The term sum_{F^ϵ} is polymorphic over other effects ε that it does not handle. The **handler** syntax defines a handler, which is a function that takes an argument of type $\mathtt{1} \to^{\mathtt{yield},\varepsilon} \mathtt{Int}$, calls this argument with unit and handles the `yield` operation. The continuation r has type $\mathtt{1} \to^{\varepsilon} \mathtt{Int}$ as it may use effects abstracted by ε. For instance, we can apply sum_{F^ϵ} as follows which reduces to 79.

$$sum_{F^\epsilon}\ E\ (\lambda x^{1}.\mathbf{do}\ \mathtt{yield}\ 42; \mathbf{do}\ \mathtt{yield}\ 37; 0)$$

We can easily encode sum_{F^ϵ} in $\textsc{Met}(\mathcal{R}_{\mathrm{scp}})$ as a polymorphic function whose body uses the **handle with** syntax to handle the argument. The main difficulty is that for the handler clause, the continuation r should have type $[\![\mathtt{1} \to^{\varepsilon} \mathtt{Int}]\!] = [\varepsilon](\mathtt{1} \to \mathtt{Int})$ following the translation of function types in Section 2.2. However, the typing rule of handlers in Tang et al. [38] only allows us to give a function type to r with no modality. To solve this problem, we introduce *modality-parameterised handlers*. In $\textsc{Met}(\mathcal{X})$, the handler syntax is annotated with a modality μ as $\mathbf{handle}^{\mu}\ M\ \mathbf{with}\ H$. The continuation r in the handler clause of H now has type $\mu(A \to B)$ for some types A and B. With the modality-parameterised handler, we can translate sum_{F^ϵ} as follows, omitting the details of the translation of the handler clause, which we name H'.

$$\begin{aligned}[\![sum_{F^\epsilon}]\!] \ &= \ \Lambda\varepsilon.\mathbf{mod}_{[\varepsilon]}\ (\lambda f^{[\mathtt{yield},\varepsilon](\mathtt{1}\to\mathtt{Int})}.\mathbf{handle}^{[\varepsilon]}\ (\mathbf{let}\ \mathbf{mod}_{[\mathtt{yield},\varepsilon]}\ f' = f\ \mathbf{in}\ f'\ ())\ \mathbf{with}\ H') \\ &: \ \forall\varepsilon.[\varepsilon]([\mathtt{yield},\varepsilon](\mathtt{1} \to \mathtt{Int}) \to \mathtt{Int})\end{aligned}$$

We eliminate the modality of the argument f before applying and handling it. The type translation follows the translation given in Section 2.2. We give full details of our modality-parameterised handlers in Section 3.5 and formally define the translation of handlers in Section 4.2.

2.5.3 Encoding Effect Handlers in System C. System C adopts named handlers. Instead of using operation labels to identify which operation we want to invoke and handle, in System C each handler binds a fresh capability in the scope of the handler and handles the use of this capability. For instance, we can define a named handler and use it to handle a computation as follows.

$$\vdash\ sum_C \doteq \mathbf{try}\ \{y^{\mathtt{Int}\Rightarrow\mathtt{1}} \Rightarrow y(42); y(37); 0\}\ \mathbf{with}\ \{p\ r \mapsto p + r(())\}\ :\ \mathtt{Int}\ \mid C$$

Handlers in System C use the **try with** syntax. This handler introduces a capability y of type $\mathtt{Int} \Rightarrow \mathtt{1}$ in the scope between **try** and **with**. We use the capability y to yield integers 42 and 37. These two uses of y are handled by the handler, whose operation clause is similar to what we have seen before, except it uses a capability in place of an operation label.

The semantics of named handlers in System C differs from that of the standard effect handlers of Plotkin and Pretnar [34]. Named handlers have a generative semantics [4] which dynamically generates a fresh runtime label for each capability introduced by a handler. Dynamic generation guarantees the uniqueness of runtime labels, which ensures that all uses of a capability must be handled by the handler that introduces the capability.

To encode the named handlers of System C into $\textsc{Met}(\mathcal{X})$, we need to resolve this semantic gap. Adding named handlers to $\textsc{Met}(\mathcal{X})$ would work but is rather heavyweight. We observe that the essence of named handlers is actually a way to dynamically generate labels. We introduce *local labels* to $\textsc{Met}(\mathcal{X})$, which decouple dynamic generation of labels from named handlers. This extension is inspired by the local effects of Biernacki et al. [3], dynamic labels of de Vilhena and Pottier [11], and

fresh labels of the Links language [19]. The syntax **local** $\ell : A \twoheadrightarrow B$ **in** M introduces a local label in the scope of M. The type system ensures the local label ℓ cannot escape from M. The semantics generates a fresh label to replace the local label ℓ. We provide the details in Section 3. With local labels, we can encode sum_C as follows, omitting the handler H', which contains an operation clause for ℓ_y translated from the handler of sum_C.

$$\begin{aligned}\vdash [\![sum_C]\!] \;=\; & \textbf{local}\ \ell_y : \mathtt{Int} \twoheadrightarrow \mathtt{1}\ \textbf{in}\ \textbf{handle}^{[[\![C]\!]]} \\ & (\lambda y^{[\ell_y](\mathtt{Int}\to\mathtt{1})}.\textbf{let}\ \mathbf{mod}_{[\ell_y]}\ \hat{y} = y\ \textbf{in}\ \hat{y}\ 42; \hat{y}\ 37; 0)\ (\mathbf{mod}_{[\ell_y]}\ (\lambda x^{\mathtt{Int}}.\textbf{do}\ \ell_y\ x))\ \textbf{with}\ H' : \mathtt{Int}\ @\ [\![C]\!]\end{aligned}$$

We introduce a local label ℓ_y for the handler. We use the term $\mathbf{mod}_{[\ell_y]}\ (\lambda x^{\mathtt{Int}}.\textbf{do}\ \ell_y\ x)$ which invokes the operation ℓ_y to simulate the capability introduced by the named handler in sum_C. The translation of the handled computation binds this function to y, eliminates the modality of y to $\hat{y}$, and uses $\hat{y}$ to yield integers 42 and 37. As in the encoding of effect handlers in System F^ϵ, we also use our modality-parameterised handlers here and annotate **handle** with the modality $[[\![C]\!]]$.

Our translation $[\![sum_C]\!]$ is simplified for clarity; it is actually the result of reducing the full translation of sum_C by a few steps. We give the full translation in Section 5.2.

2.5.4 Comparing Encodings of Effect Handlers. Our encodings of System F^ϵ and System C effect handlers elucidate how effect handlers differ in these two effect systems.

(1) The System C encoding requires local labels, whereas the System F^ϵ encoding does not, which reveals the syntactic difference that capabilities in System C have scopes whereas operation labels in System F^ϵ do not, and the semantic difference that System C generates fresh runtime labels for effect handlers, whereas System F^ϵ does not.
(2) The System F^ϵ encoding performs operations directly, whereas the System C encoding wraps operation invocations into a function (such as the term $\mathbf{mod}_{[\ell_y]}(\lambda x^{\mathtt{Int}}.\textbf{do}\ \ell_y\ x)$ in $[\![sum_C]\!]$) and passes this function to the handled computation. This difference shows how in a capability-based effect system such as System C operations are not directly invoked via their labels but are instead invoked and passed around as blocks explicitly at the term level.

2.6 More Encodings

The encodings of System F^ϵ and System C illustrate the core idea of using modalities to encode and compare effect systems with different foundations in $\textsc{Met}(\mathcal{X})$. However, $\textsc{Met}(\mathcal{X})$ can be used for much more than encoding these two effect systems. In Section 6, we will discuss two more representative encodings of effect systems into $\textsc{Met}(\mathcal{X})$, including

- System Ξ [7], an early core calculus for Effekt based on capabilities, and
- System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ [41], a core calculus formalising scope-safe named handlers of Koka.

These results further demonstrate the expressiveness of $\textsc{Met}(\mathcal{X})$ as a general framework to encode, compare, and analyse effect systems. We further discuss practical language design insights arising from our encodings in Section 6.3.

3 The Core Calculus $\textsc{Met}(\mathcal{X})$

$\textsc{Met}(\mathcal{X})$ is a System F-style call-by-value core calculus with modal effect types parameterised by an effect structure $\mathcal{X}$. In addition to the effect structure, $\textsc{Met}(\mathcal{X})$ also extends $\textsc{Met}$ with local labels and modality-parameterised handlers. We aim to be self-contained about modal effect types in this paper and refer to Tang et al. [38] for a more complete introduction.

3.1 Syntax

The syntax of $\textsc{Met}(\mathcal{X})$ is as follows. We highlight syntax relevant to modal effect types and our extensions of local labels and modality-parameterised handlers in grey.

$$
\begin{array}{lll}
\text{Types} & A, B ::= \mathsf{1} \mid A \to B \mid \mu A & \\
& \quad \mid \alpha \mid \forall \alpha^K.A & \\
\text{Modalities} & \mu, \nu ::= [E] \mid \langle D \rangle & \\
\text{Extensions} & D ::= \cdot \mid \ell, D \mid \varepsilon, D & \\
\text{Effect Contexts} & E, F ::= \cdot \mid \ell, E \mid \varepsilon, E & \\
\text{Kinds} & K ::= \mathsf{Abs} \mid \mathsf{Any} \mid \mathsf{Effect} & \\
\text{Contexts} & \Gamma ::= \cdot \mid \Gamma, \alpha : K \mid \Gamma, \blacksquare_{\mu_E} & \\
& \quad \mid \Gamma, x :_{\mu_E} A \mid \Gamma, \ell : A \twoheadrightarrow B & \\
\text{Label Contexts} & \Sigma ::= \cdot \mid \Sigma, \ell : A \twoheadrightarrow B &
\end{array}
$$

$$
\begin{array}{lll}
\text{Terms} & M, N ::= () \mid x \mid \lambda x^A.M \mid M\,N & \\
& \quad \mid \Lambda \alpha^K.V \mid M\,A \mid \mathbf{mod}_\mu\, V & \\
& \quad \mid \mathbf{let}_\nu\, \mathbf{mod}_\mu\, x = V\ \mathbf{in}\ M & \\
& \quad \mid \mathbf{do}\ \ell\ M \mid \mathbf{local}\ \ell : A \twoheadrightarrow B\ \mathbf{in}\ M & \\
& \quad \mid \mathbf{handle}^\mu\ M\ \mathbf{with}\ H & \\
\text{Values} & V, W ::= () \mid x \mid \lambda x^A.M \mid \Lambda \alpha^K.V \mid \mathbf{mod}_\mu\, V & \\
& \quad \mid V\,A \mid \mathbf{let}_\nu\, \mathbf{mod}_\mu\, x = V\ \mathbf{in}\ W & \\
\text{Handlers} & H ::= \{\mathbf{return}\ x \mapsto N, \ell\ p\ r \mapsto M\} &
\end{array}
$$

We have two kinds Abs and Any for value types and one kind Effect for extensions and effect contexts. By convention, we usually write α for type variables of values and ε for effect variables. We omit kinds when obvious. We let A range over both value types A and effect contexts E, and let α range over type variables for them in type abstraction $\Lambda \alpha^K.V$ and type application $M\,A$.

Unlike Tang et al. [38], we omit masking, as it is not used by our encodings. We discuss future extensions to $\textsc{Met}(\mathcal{X})$, including masking, in Section 6.4.

For simplicity, we assume that each handler only handles one operation, and fix a global context Σ which associates each global operation label with its type. An entry $\ell : A \twoheadrightarrow B$ indicates that the operation ℓ takes an argument of type A and returns a value of type B. We also support local labels which are introduced by $\mathbf{local}\ \ell : A \twoheadrightarrow B\ \mathbf{in}\ M$ and maintained in the context Γ. We do not distinguish between local and global labels syntactically.

Values include type application and modality elimination whose subterms are restricted to be values, following the notion of *complex values* in call-by-push-value [26]. Such complex values are convenient as we adopt a value restriction [39] for type abstraction and modality introduction.

3.2 Effect Structures

An effect structure defines the structure of effect collections, that is, extensions and effect contexts in $\textsc{Met}(\mathcal{X})$. Extensions D and effect contexts E are both syntactically defined as lists of labels and effect variables. We overload commas for list concatenation, e.g., D, E and E, F are both list concatenation. As usual, list concatenation is associative but not commutative. The kinding, equivalence, and subtyping (or subeffecting) relations for them are determined by an effect structure $\mathcal{X}$.

Definition 3.1 (Effect structure). An effect structure $\mathcal{X}$ is a tuple $\langle :, \equiv \rangle$ of two relations.

- $\Gamma \vdash D : \mathsf{Effect}$ is a kinding relation which defines well-formed extensions and is preserved by concatenation D, D'. That is, if $\Gamma \vdash D : \mathsf{Effect}$ and $\Gamma \vdash D' : \mathsf{Effect}$, then $\Gamma \vdash D, D' : \mathsf{Effect}$.
- $\Gamma \vdash D \equiv D'$ is an equivalence relation for well-formed extensions.

Our definition of an effect structure $\mathcal{X}$ is minimal and only includes the definitions of kinding and equivalence relations for extensions D. We can naturally derive the kinding relation $\Gamma \vdash E : \mathsf{Effect}$, equivalence relation $\Gamma \vdash E \equiv E'$, and subeffecting relation $\Gamma \vdash E \leqslant E'$ for effect contexts as follows.

$$
\frac{}{\Gamma \vdash \cdot : \mathsf{Effect}} \qquad \frac{\Gamma \ni \varepsilon : \mathsf{Effect}}{\Gamma \vdash \varepsilon : \mathsf{Effect}} \qquad \frac{\Gamma \vdash D : \mathsf{Effect} \qquad \Gamma \vdash E : \mathsf{Effect}}{\Gamma \vdash D, E : \mathsf{Effect}}
$$

$$
\frac{}{\Gamma \vdash \cdot \equiv \cdot} \qquad \frac{}{\Gamma \vdash \varepsilon \equiv \varepsilon} \qquad \frac{\Gamma \vdash D_1 \equiv D_2 \qquad \Gamma \vdash E_1 \equiv E_2}{\Gamma \vdash D_1, E_1 \equiv D_2, E_2} \qquad \frac{\Gamma \vdash E, E' \equiv F}{\Gamma \vdash E \leqslant F}
$$

The kinding and equivalence relations for effect contexts are defined inductively. The subeffecting relation is more interesting. We have $E \leqslant F$ if there exists an effect context E' such that E, E' is well-formed and $E, E' \equiv F$. It is easy to verify that this subeffecting relation is a preorder. We often write $:_{\mathcal{X}}$, $\leqslant_{\mathcal{X}}$, and $\equiv_{\mathcal{X}}$ to denote which specific effect structure we refer to. We sometimes omit the context Γ for the equivalence and subeffecting for brevity.

We give three examples of effect structures, among which $\mathcal{R}_{\text{scp}}$ and $\mathcal{S}$ are used for the encoding of System F^{ϵ} and System C in Sections 4.2 and 5.2, respectively.

Definition 3.2 (Simple Rows). $\mathcal{R}_{\text{simp}} = \langle :_{\mathcal{R}_{\text{simp}}}, \equiv_{\mathcal{R}_{\text{simp}}} \rangle$ defines effect collections as simple rows [30] of operation labels. Well-formed extensions consist of distinct labels without any effect variable. $D \equiv D'$ if D is identical to D' modulo reordering of labels.

Definition 3.3 (Scoped Rows). $\mathcal{R}_{\text{scp}} = \langle :_{\mathcal{R}_{\text{scp}}}, \equiv_{\mathcal{R}_{\text{scp}}} \rangle$ defines effect collections as scoped rows [23] of operation labels. Well-formed extensions comprise potentially duplicated labels without any effect variable. $D \equiv D'$ if D is identical to D' modulo reordering of distinct labels.

Definition 3.4 (Sets). $\mathcal{S} = \langle :_{\mathcal{S}}, \equiv_{\mathcal{S}} \rangle$ defines effect collections as sets. Well-formed extensions are sets of labels and effect variables. The equivalence relation is set equivalence.

Full formal definitions of these effect structures are given in **??**. The effect structure $\mathcal{R}_{\text{scp}}$ corresponds to the treatment of effect collections as scoped rows used in Eff, modulo the fact that Eff has presence types for labels in effect contexts, whereas we choose not to for simplicity. We discuss extending Eff($\mathcal{X}$) with presence types and richer effect kinds in Section 6.4.

Following Yoshioka et al. [43], an effect structure that intuitively characterises the notion of a collection of effects should satisfy the following validity conditions.

Definition 3.5 (Validity Conditions). Validity conditions for an effect structure $\mathcal{X}$ are

(1) if $E \leqslant_{\mathcal{X}} \cdot$ then $E = \cdot$, and
(2) if $\ell \leqslant_{\mathcal{X}} \ell', E$ and $\ell \neq \ell'$ then $\ell \leqslant_{\mathcal{X}} E$.

The validity conditions together ensure that if a label ℓ is a subtype of an effect context E, then it must syntactically appear in the effect context E. The first condition prevents us from claiming that some label is contained in the empty effect context. The second condition prevents us from identifying two syntactically different label as the same one. All effect structures given above satisfy the validity conditions. Our type soundness and effect safety theorems in Section 3.7 are parameterised by any effect structure satisfying the validity conditions.

3.3 Modalities

Modalities manipulate effect contexts as follows.

$$[E](F) = E \qquad\qquad \langle D \rangle(F) = D, F$$

The absolute modality $[E]$ completely replaces the effect context F with E. The extension modality $\langle D \rangle$ extends the effect context F with D. Following Eff [38], we write μ_F as a meta-level notation for the pair of modality μ and effect context F where F is the effect context that μ manipulates.

Modality Composition. We define the composition of modalities as follows.

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

Composition is from left to right, for consistency with Eff. First, an absolute modality fully determines the new effect context E no matter what μ does before. Second, setting the effect context to E followed by extending E with D is equivalent to directly setting the effect context to D, E. Third, relative modalities can be composed into one by combining the extensions. Composition is

well-defined as we have $(\mu \circ \nu)(E) = \nu(\mu(E))$. We also have associativity $(\mu \circ \nu) \circ \xi = \mu \circ (\nu \circ \xi)$ and identity $\langle\rangle \circ \mu = \mu \circ \langle\rangle = \mu$. All of these properties are independent of the effect structure $\mathcal{X}$.

Modality Transformation. We define a modality transformation judgement, which determines the coercion of modalities, controlling the accessibility of variables as mentioned in Section 2.1.2 where we disallow the usage of the variable $f : \texttt{Int} \to \texttt{1}$. Given a variable binding $f :_{\mu_F} A$ (which means f is introduced by eliminating the modality μ of some value of type μA at effect context F), we can access it after a lock $\blacksquare_{\nu_F}$ if the modality transformation relation $\Gamma \vdash \mu \Rightarrow \nu \mathbin{@} F$ holds. The modality transformation judgement is defined as follows.

$$\textsc{MT-Abs}\ \frac{\Gamma \vdash E \leqslant \mu(F)}{\Gamma \vdash [E] \Rightarrow \mu \mathbin{@} F} \qquad \textsc{MT-Extend}\ \frac{\Gamma \vdash D_1, F \leqslant D_2, F \text{ for all } E \leqslant F}{\Gamma \vdash \langle D_1\rangle \Rightarrow \langle D_2\rangle \mathbin{@} E}$$

Both rules make sure that we do not lose any effects after transformation. Rule MT-Abs allows us to transform an absolute modality $[E]$ to any other modality μ as long as $E \leqslant \mu(F)$. Rule MT-Extend allows us to transform an extension modality $\langle D_1\rangle$ to another extension modality $\langle D_2\rangle$ as long as for any effect context F larger than E, we have $D_1, F \leqslant D_2, F$. We need to quantify over all effect contexts F which are larger than the ambient effect context E because the new effect context that a relative modality gives us depends on the ambient effect context. For instance, consider the following judgement which coerces the modality $\langle D_1\rangle$ of V to $\langle D_2\rangle$.

$$\Gamma \vdash \textbf{let mod}_{\langle D_1\rangle}\ x = V\ \textbf{in mod}_{\langle D_2\rangle}\ x : \langle D_2\rangle(\texttt{Int} \to \texttt{Int}) \mathbin{@} E$$

In its derivation tree we need the transformation relation $\langle D_1\rangle \Rightarrow \langle D_2\rangle \mathbin{@} E$. To preserve the judgement after upcasting E to a larger F, the transformation requires $D_1, F \leqslant D_2, F$ for any $E \leqslant F$.

Our MT-Extend rule is suitable for any effect structure, while the corresponding rule MT-Upcast in Tang et al. [38] is specific to the treatment of effect collections as scoped rows in Met. Given a specific effect structure, we can usually find an easier-to-compute representation of MT-Extend without universal quantification (as is the case for the MT-Upcast rule in Met).

3.4 Kinds and Contexts

The kinding relations for extensions and effect contexts are provided by the effect structure $\mathcal{X}$ in Section 3.2. For value types, we have two kinds where Abs is a subkind of Any. A type has kind Abs if all function types appearing as syntactic subterms of the type are wrapped in absolute modalities. For example, $(\texttt{1} \to \texttt{1}) \to \texttt{1}$ does not have kind Abs whereas $[](\texttt{1} \to \texttt{1}) \to \texttt{1})$ does. Intuitively, values whose types have kind Abs do not depend on the ambient effect context. For any operation $\ell : A \twoheadrightarrow B$, types A and B should have kind Abs to avoid effect leakage following Tang et al. [38]. The kinding and type equivalence rules of $\textsc{Met}(\mathcal{X})$ are given in **??**.

Contexts are ordered. We write $\Gamma \mathbin{@} E$ when context Γ is well-formed at effect context E, that is, the types of the variables are well-kinded, and the variables and locks are compatible with E. For instance, the following context is well-formed at effect context E.

$$x :_{\mu_F} A_1, y :_{\nu_F} A_2, \blacksquare_{[E]_F}, z :_{\xi_E} A_3, w : A_4 \mathbin{@} E$$

Let us read from right to left. Variable w is at effect context E (it is technically tagged with an identity modality $\langle\rangle_E$ which is omitted). Variable z is tagged with modality ξ_E, which means it is not at effect context E but actually at effect context $\xi(E)$. Lock $\blacksquare_{[E]_F}$ changes the effect context to E from F. Variables y and x are at effect contexts $\nu(F)$ and $\mu(F)$, respectively. Each modality in the context carries an index of the effect context it manipulates, making switching of effect contexts explicit. We frequently omit the index when it is clear what it must be. Formal definitions of kinding and context well-formedness rules are in **??**. We define locks$(-)$ to compose all the modalities on

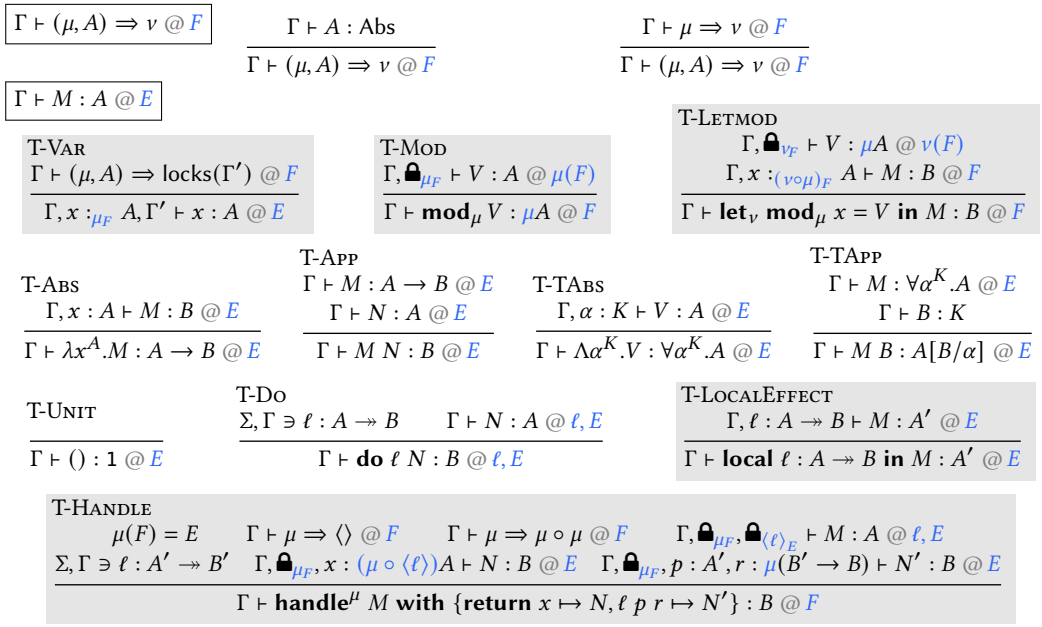


Fig. 1. Typing rules and auxiliary rules of Met($\mathcal{X}$).

the locks in a context.

$$\mathrm{locks}(\cdot) = \langle\rangle \qquad \mathrm{locks}(\Gamma, \text{🔒}_{\mu_E}) = \mathrm{locks}(\Gamma) \circ \mu \qquad \mathrm{locks}(\Gamma, x :_{\mu_E} A) = \mathrm{locks}(\Gamma)$$

We identify contexts up to the following two equations.

$$\Gamma, \text{🔒}_{\langle\rangle_F}, \Gamma' \ @\ E = \Gamma, \Gamma' \ @\ E \qquad\qquad \Gamma, \text{🔒}_{\mu_F}, \text{🔒}_{\nu_{F'}}, \Gamma' \ @\ E = \Gamma, \text{🔒}_{(\mu \circ \nu)_F}, \Gamma' \ @\ E$$

3.5 Typing

Figure 1 gives the typing rules for Met($\mathcal{X}$). As before, we highlight rules relevant to modal effect types and our extensions in grey. The typing judgement $\Gamma \vdash M : A \ @\ E$ means that the term M has type A under context Γ and effect context E with well-formedness condition $\Gamma \ @\ E$.

Modality Introduction and Elimination. Rule T-Mod introduces a modality μ to the conclusion, puts a lock into the context of the premise, and changes the effect context. Rule T-Letmod eliminates a modality μ and moves it to the variable binding. We have seen examples that rely on these rules in Section 2.1. There is another modality ν in T-Letmod which is needed for technical reasons to support sequential elimination. For instance, given a variable $x : \nu\mu A$ with two modalities, to eliminate both ν and μ, we can first eliminate ν to $y :_\nu \mu A$ and then to $z :_{\nu\circ\mu} A$ as follows.

$$\mathbf{let}\ \mathbf{mod}_\nu\ y = x\ \mathbf{in}\ \mathbf{let}_\nu\ \mathbf{mod}_\mu\ z = y\ \mathbf{in}\ M$$

We restrict $\mathbf{mod}_\mu$ and $\mathbf{let}_\nu\ \mathbf{mod}_\mu$ to values to avoid effect leakage, as in Met [29, 38]. Otherwise, for example, if we were to allow a computation $\mathbf{mod}_{[\mathtt{yield}]}\ (\mathbf{do}\ \mathtt{yield}\ 42)$, this term would be well-typed in the empty effect context but get stuck as `yield` is not handled (note that we do not want **mod** to suspend computations as it could be confusing to programmers).

Accessing Variables. Locks control the accessibility of variables as we have shown in Section 2.1. Rule T-Var uses the auxiliary judgement $\Gamma \vdash (\mu, A) \Rightarrow \mathsf{locks}(\Gamma')\ @\ F$ (also defined in Figure 1) to check whether we can access a variable $x :_{\mu_F} A$ given all locks in Γ'. When A has kind Abs, we can always use x as it does not depend on the effect context. Otherwise we need to make sure the coercion from μ to $\mathsf{locks}(\Gamma')$ is safe by checking the modality transformation relation $\Gamma \vdash \mu \Rightarrow \mathsf{locks}(\Gamma')\ @\ F$ where $\mathsf{locks}(\Gamma')$ composes the modalities on locks in Γ'. We have seen an example in Section 2.1.2 that the variable $f : \mathtt{Int} \to \mathtt{1}$ cannot be used while $f :_{[\mathtt{yield}]} \mathtt{Int} \to \mathtt{1}$ can. As another example, $x :_{\langle \ell \rangle} \mathtt{1} \to \mathtt{1}, \text{🔒}_{[\ell']} \vdash x : \mathtt{1} \to \mathtt{1}\ @\ \ell'$ is ill-typed since we cannot transform the modality $\langle \ell \rangle$ to $[\ell']$. It would be well-typed if x had type $\mathtt{Int}$.

Local Labels. Rule T-LocalEffect binds a fresh local label ℓ with type signature $A \twoheadrightarrow B$ (we adopt the Barendregt convention for local labels.) Well-formedness of type A' and effect context E under Γ ensures that ℓ cannot appear in A' or E. Rule T-Do may use any label from Σ and Γ. The operational semantics (Section 3.6) generates runtime labels to substitute local labels.

Modality-Parameterised Handlers. Rule T-Handle defines a handler and uses it to handle a computation M. Let us first ignore all occurrences of the modality μ. A handler of operation ℓ extends the effect context with ℓ as indicated by the lock $\text{🔒}_{\langle \ell \rangle_E}$ in the typing judgement of M. The return value of M is bound to the variable x in the return clause $\mathbf{return}\ x \mapsto N$. The type of x also has the modality $\langle \ell \rangle$ since x may use the operation ℓ, e.g., when M returns a function $\lambda x.\mathbf{do}\ \ell\ x$.

We generalise the handlers of Tang et al. [38] to be parameterised by a modality μ. The modality μ transforms the effect context F to $\mu(F) = E$ for the whole term as witnessed by the addition of the lock 🔒_{μ_F} to the context of each premise. Since both the handled computation and the handler clauses are well-typed under the lock 🔒_{μ_F}, we can wrap the continuation r, which captures the handled computation and the handler, into the modality μ. The return value x is also wrapped in the modality μ as it is returned from M whose context contains the lock 🔒_{μ_F}. This is in contrast to the handler rule of Tang et al. [38], as shown below, which just gives r the function type $B' \to B$.

$$\frac{\Sigma, \Gamma \ni \ell : A' \twoheadrightarrow B' \quad \Gamma, \text{🔒}_{\langle \ell \rangle_E} \vdash M : A\ @\ \ell, E \quad \Gamma, x : \langle \ell \rangle A \vdash N : B\ @\ E \quad \Gamma, p : A', r : B' \to B \vdash N' : B\ @\ E}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N, \ell\ p\ r \mapsto N'\} : B\ @\ E}$$

To recover the original handler construct of Tang et al. [38], we just need to instantiate the modality μ to the identity modality $\langle\rangle$ as shown by the following syntactic sugar.

$$\begin{aligned} & \mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N, \ell\ p\ r \mapsto N'\} \\ \doteq\ & \mathbf{handle}^{\langle\rangle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N, \ell\ p\ r \mapsto \mathbf{let}\ \mathbf{mod}_{\langle\rangle}\ r = r\ \mathbf{in}\ N'\} \end{aligned}$$

Having a modality μ for the continuation r allows us to have more fine-grained control over effect tracking for the continuation. As discussed in Section 2.5, the extra expressiveness provided by this rule is especially useful for a unified framework to encode other effect systems with support for effect handlers, as different encodings typically require translating a function type into a type with some modalities. We give an example of an effect handler annotated with the empty absolute modality $[]$ in $\textsc{Met}(\mathcal{X})$ based on the handler $sum_{\textsc{Met}(\mathcal{X})}$ in Section 2.5.1.

$$\begin{aligned} \mathbf{handle}^{[]}\ (\mathbf{do}\ \mathtt{yield}\ 42; \mathbf{do}\ \mathtt{yield}\ 37; 0)\ \mathbf{with}\ \{&\mathbf{return}\ x \mapsto \mathbf{let}\ \mathbf{mod}_{[\mathtt{yield}]}\ x' = x\ \mathbf{in}\ x', \\ &\mathtt{yield}\ p\ r \mapsto \mathbf{let}\ \mathbf{mod}_{[]}\ r' = r\ \mathbf{in}\ p + r'\ ()\} \end{aligned}$$

As a result of the annotation $[]$, the continuation r has type $[](\mathtt{1} \to \mathtt{Int})$ instead of $\mathtt{1} \to \mathtt{Int}$. In the return clause we eliminate the modality $[] \circ \langle \mathtt{yield} \rangle = [\mathtt{yield}]$ of x. In contrast, the omitted return clause of $sum_{\textsc{Met}(\mathcal{X})}$ is $\mathbf{return}\ x \mapsto \mathbf{let}\ \mathbf{mod}_{\langle \mathtt{yield} \rangle}\ x' = x\ \mathbf{in}\ x'$.

The new handler rule requires the modality μ to have a comonadic structure as specified by the conditions $\Gamma \vdash \mu \Rightarrow \langle\rangle\ @\ F$ and $\Gamma \vdash \mu \Rightarrow \mu \circ \mu\ @\ F$. These conditions are important because

$$\begin{array}{lll} \text{Value normal forms} & U ::= x \mid \lambda x^{A}.M \mid \Lambda\alpha^{K}.V \mid \mathbf{mod}_{\mu}\, U \\ \text{Evaluation Contexts} & \mathcal{E} ::= [\,] \mid \mathcal{E}\, N \mid U\, \mathcal{E} \mid \mathcal{E}\, A \mid \mathbf{mod}_{\mu}\, \mathcal{E} \mid \mathbf{let}_{\nu}\, \mathbf{mod}_{\mu}\, x = \mathcal{E}\, \mathbf{in}\, M \\ & \quad \mid \mathbf{do}\, \ell\, \mathcal{E} \mid \mathbf{handle}^{\mu}\, \mathcal{E}\, \mathbf{with}\, H \end{array}$$

$$\begin{array}{lll} \text{E-App} & (\lambda x^{A}.M)\, U \rightsquigarrow M[U/x] & \\ \text{E-TApp} & (\Lambda\alpha^{K}.U)\, A \rightsquigarrow U[A/\alpha] & \\ \text{E-Letmod} & \mathbf{let}_{\nu}\, \mathbf{mod}_{\mu}\, x = \mathbf{mod}_{\mu}\, U\, \mathbf{in}\, M \rightsquigarrow M[U/x] & \\ \text{E-Gen} & \mathbf{local}\, \ell : A \twoheadrightarrow B\, \mathbf{in}\, M \mid \Omega \rightsquigarrow M[\ell'/\ell] \mid \Omega, \ell' : A \twoheadrightarrow B & \text{where } \ell' \text{ fresh in } \Omega \text{ and } \Sigma \\ \text{E-Ret} & \mathbf{handle}^{\mu}\, U\, \mathbf{with}\, H \rightsquigarrow N[(\mathbf{mod}_{(\mu \circ \langle \ell \rangle)}\, U)/x], & \\ & \qquad \text{where } H = \{\mathbf{return}\, x \mapsto N, \ell\ p\ r \mapsto N'\} & \\ \text{E-Op} & \mathbf{handle}^{\mu}\, \mathcal{E}[\mathbf{do}\, \ell\, U]\, \mathbf{with}\, H \rightsquigarrow N[U/p, (\mathbf{mod}_{\mu}\, (\lambda y.\mathbf{handle}^{\mu}\, \mathcal{E}[y]\, \mathbf{with}\, H))/r] & \\ & \qquad \text{where } \ell \notin \mathsf{bl}(\mathcal{E}) \text{ and } H \ni (\ell\ p\ r \mapsto N) & \\ \text{E-Lift} & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N] & \text{if } M \rightsquigarrow N \end{array}$$

Fig. 2. Operational semantics of Met($\mathcal{X}$).

semantically a handler for operation ℓ may not be used (when ℓ is not invoked) or be used multiple times (when ℓ is invoked multiple times). Intuitively, each use of the handler consumes one modality μ. The condition $\mu \Rightarrow \langle\rangle \ @\ F$ makes sure that when the handler is not used, we can transform away the modality μ at effect context F. The condition $\mu \Rightarrow \mu \circ \mu \ @\ F$ makes sure that when the handler is used multiple times, we can duplicate the modality μ each time the handlers is used. For example, the identity modality $\langle\rangle$ trivially satisfies the comonadic structure, and the absolute modality $[E]$ satisfies the comonadic structure at F with $E \leqslant F$.

3.6 Operational Semantics

We adopt the generative semantics of Biernacki et al. [4] for local labels. Each local label ℓ introduced by $\mathbf{local}\ \ell : A \twoheadrightarrow B\ \mathbf{in}\ M$ is replaced by a fresh label generated at runtime. We manage these labels in a context defined as $\Omega ::= \cdot \mid \Omega, \ell : A \twoheadrightarrow B$. We do not syntactically distinguish runtime generated labels from static labels; runtime labels are tracked in Ω. We define value normal forms U which cannot reduce further. The definitions for all new syntax and the operational semantics are given in Figure 2. The reduction relation has the form $M \mid \Omega \rightsquigarrow N \mid \Omega'$. We omit Ω when it is unchanged. Only E-Gen extends Ω. We do not restrict M and N to be closed terms. All judgements defined previously are also straightforwardly extended with Ω. For instance, typing judgements are of form $\Omega \mid \Gamma \vdash M : A \ @\ E$ for runtime terms.

The operational semantics mostly follows Met. Rule E-Gen is new and generates a fresh runtime label for a local label binding. Moreover, since we generalise the handler of Met, rules E-ret and E-Op are also generalised. Rule E-Ret wraps the return value with the modality $\mu \circ \langle \ell \rangle$. Rule E-Op wraps the continuation with the modality μ. The modalities in both rules are consistent with the typing rule T-Handle in Section 3.5. The function $\mathsf{bl}(\mathcal{E})$ gives the set of bound operation labels which have handlers installed in the evaluation context $\mathcal{E}$. The condition $\ell \notin \mathsf{bl}(\mathcal{E})$ makes sure each operation ℓ is handled by the dynamically innermost handler of ℓ.

3.7 Type Soundness and Effect Safety

To state syntactic type soundness, we first define normal forms.

Definition 3.6 (Normal Forms). We say that term M is in normal form with respect to effect context E, if it is either in value normal form $M = U$ or of the form $M = \mathcal{E}[\mathbf{do}\, \ell\, U]$ for $\ell \leqslant E$.

The following theorems together give type soundness and effect safety. They hold for any effect structure $\mathcal{X}$ satisfying the validity conditions of Definition 3.5.

Theorem 3.7 (Progress). *In Met($\mathcal{X}$) where $\mathcal{X}$ satisfies the validity conditions, if $\Omega \mid \cdot \vdash M : A \mathbin{@} E$, then either $M \mid \Omega \rightsquigarrow N \mid \Omega'$ for some N and Ω', or M is in a normal form with respect to E.*

Theorem 3.8 (Subject Reduction). *In Met($\mathcal{X}$) where $\mathcal{X}$ satisfies the validity conditions, if $\Omega \mid \Gamma \vdash M : A \mathbin{@} E$ and $M \mid \Omega \rightsquigarrow N \mid \Omega'$, then $\Omega' \mid \Gamma \vdash N : A \mathbin{@} E$.*

The proofs are given in **??**.

4 Encoding a Row-Based Effect System à la Koka

In this section, we briefly present System F^ϵ [40], a System F-style core calculus formalising the row-based effect system of Koka [25], and show how to encode it into Met($\mathcal{R}_{\text{scp}}$). We refer to Xie et al. [40] for a complete introduction to System F^ϵ.

4.1 System F^ϵ

The syntax of System F^ϵ is as follows.

Value Types	$A, B ::= 1 \mid \alpha \mid A \to^E B \mid \forall \alpha^K.A$	Values	$V, W ::= () \mid x \mid \lambda^E x^A.M$
Effect Rows	$E, F ::= \cdot \mid \varepsilon \mid \ell, E$		$\mid \Lambda \alpha^K.V \mid \mathbf{handler}\ H$
Kind	$K ::= \text{Effect} \mid \text{Value}$	Computations	$M, N ::= \mathbf{return}\ V \mid V\ W \mid V\ A$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha : K$		$\mid \mathbf{do}\ \ell\ V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N$
Label Contexts	$\Sigma ::= \cdot \mid \Sigma, \ell : A \twoheadrightarrow B$	Handlers	$H ::= \{\ell\ p\ r \mapsto N\}$

Different from Xie et al. [41], our version of System F^ϵ is fine-grain call-by-value [27]. Effect rows E are scoped rows [23] with an optional tail effect variable ε. As in Met($\mathcal{X}$), we assume a fixed global label context Σ. By convention we write ε for effect variables and α for value type variables. We omit their kinds, Effect and Value, when obvious. In type abstraction and application, we let A range over both value types and effect rows, and let α range over their type variables.

Typing judgements in System F^ϵ include $\Gamma \vdash V : A$ for values and $\Gamma \vdash M : A \mid E$ for computations, where the latter tracks effects E. The typing rules and operational semantics of System F^ϵ are standard for a System F-style calculus with effect handlers and a row-based effect system [16, 24]. We provide the full rules in **??** and show three representative typing rules here.

$$\text{T-Abs}\ \frac{\Gamma, x : A \vdash M : B \mid E}{\Gamma \vdash \lambda^E x^A.M : A \to^E B} \qquad \text{T-Do}\ \frac{\Sigma \ni \ell : A \twoheadrightarrow B \quad \Gamma \vdash V : A}{\Gamma \vdash \mathbf{do}\ \ell\ V : B \mid \ell, E}$$

$$\text{T-Handler}\ \frac{H = \{\ell\ p\ r \mapsto N\} \quad \Sigma \ni \ell : A' \twoheadrightarrow B' \quad \Gamma, p : A', r : B' \to^E A \vdash N : A \mid E}{\Gamma \vdash \mathbf{handler}\ H : (1 \to^{\ell,E} A) \to^E A}$$

Rule T-Abs introduces a λ-abstraction. Rule T-Do invokes an operation ℓ. Rule T-Handler introduces a handler as a function that takes an argument function of type $1 \to^{\ell,E} A$ as in Section 2.5.2. Xie et al. [40] do not include a return clause in handlers for System F^ϵ.

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

Figure 3 encodes System F^ϵ in Met($\mathcal{R}_{\text{scp}}$). The translation is mostly straightforward.

For kinds, we translate effect kind Effect to effect kind Effect and value kind Value to the kind Abs. We always translate values in System F^ϵ into values of kind Abs in Met($\mathcal{R}_{\text{scp}}$).

For types, we decouple effects from function types in System F^ϵ by translating an effectful function type $A \to^E B$ into a function type with an absolute modality $[\![E]\!]([\![A]\!] \to [\![B]\!])$.

For contexts, we homomorphically translate each entry.

For terms, the translation is type-directed and essentially defined on typing judgements. We annotate components of a term with their types as necessary. We highlight modality-relevant syntax of the term translation in grey. The grey parts show how modalities decouple effect tracking. The black parts themselves remain valid programs after type erasure.

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Kind} \to \text{Kind} \\
[\![\text{Effect}]\!] & = & \text{Effect} \\
[\![\text{Value}]\!] & = & \text{Abs}
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Effect Row} \to \text{Effect Context} \\
[\![\cdot]\!] & = & \cdot \\
[\![\varepsilon]\!] & = & \varepsilon \\
[\![\ell, E]\!] & = & \ell, [\![E]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Label Context} \to \text{Label Context} \\
[\![\cdot]\!] & = & \cdot \\
[\![\Sigma, \ell : A \twoheadrightarrow B]\!] & = & [\![\Sigma]\!], \ell : [\![A]\!] \twoheadrightarrow [\![B]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Value / Handler} \to \text{Term} \\
[\![()]\!] & = & () \\
[\![x]\!] & = & x \\
[\![\Lambda\alpha^K.V]\!] & = & \Lambda\alpha^{[\![K]\!]}.[\![V]\!] \\
[\![\lambda^E x^A.M]\!] & = & \mathbf{mod}_{[\![E]\!]}\ (\lambda x^{[\![A]\!]}.[\![M]\!]) \\
\left[\!\!\left[\begin{array}{l}\textbf{handler}\ H : \\ (1 \to^{\ell,E} A) \to^E A\end{array}\right]\!\!\right] & = & \mathbf{mod}_{[\![E]\!]}\ (\lambda f.\textbf{handle}^{[[\![E]\!]]}\ (\textbf{let}\ \mathbf{mod}_{[[\![\ell,E]\!]]}\ f' = f\ \textbf{in}\ f'\ ())\ \textbf{with}\ [\![H^E]\!]) \\
[\![\{\ell\ p\ r \mapsto N\}^E]\!] & = & \{\textbf{return}\ x \mapsto \textbf{let}\ \mathbf{mod}_{[[\![\ell,E]\!]]}\ x' = x\ \textbf{in}\ x', \ell\ p\ r \mapsto [\![N]\!]\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Type} \to \text{Type} \\
[\![1]\!] & = & 1 \\
[\![\alpha]\!] & = & \alpha \\
[\![A \to^E B]\!] & = & [[\![E]\!]]([\![A]\!] \to [\![B]\!]) \\
[\![\forall\alpha^K.A]\!] & = & \forall\alpha^{[\![K]\!]}.[\![A]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Context} \to \text{Context} \\
[\![\cdot]\!] & = & \cdot \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], x : [\![A]\!] \\
[\![\Gamma, \alpha : K]\!] & = & [\![\Gamma]\!], \alpha : [\![K]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Computation} \to \text{Term} \\
[\![\textbf{return}\ V]\!] & = & [\![V]\!] \\
[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] & = & \textbf{let}\ x = [\![M]\!]\ \textbf{in}\ [\![N]\!] \\
[\![V\ A]\!] & = & [\![V]\!]\ [\![A]\!] \\
[\![(V : A \to^E B)\ W]\!] & = & \textbf{let}\ \mathbf{mod}_{[[\![E]\!]]}\ x = [\![V]\!]\ \textbf{in}\ x\ [\![W]\!] \\
[\![\textbf{do}\ \ell\ V]\!] & = & \textbf{do}\ \ell\ [\![V]\!]
\end{array}
$$

Fig. 3. An encoding of System F$^\epsilon$ in Met($\mathcal{R}_{\text{scp}}$).

The translation of lambda abstraction $\lambda^E x^A.M$ introduces an absolute modality by $\mathbf{mod}_{[[\![E]\!]]}$, and the translation of function application $V\ W$ first eliminates the modality of $[\![V]\!]$ by $\textbf{let}\ \mathbf{mod}_{[[\![E]\!]]}\ x = [\![V]\!]$ before applying it. Examples for translations of lambda abstraction and application can be found in Section 2.2 as $[\![gen_{\mathsf{F}^\epsilon}]\!]$ and $[\![app_{\mathsf{F}^\epsilon}]\!]$.

Translations of type abstraction, type application, operation invocation, and let-binding are homomorphic. Let-binding in Met($\mathcal{X}$) is syntactic sugar defined in the standard way as $\textbf{let}\ x = M\ \textbf{in}\ N \doteq (\lambda x.N)\ M$. The translation of $\textbf{return}\ V$ is simply $[\![V]\!]$.

A handler value $\textbf{handler}\ H$ of type $(1 \to^{\ell,E} A) \to^E A$ is translated to a higher-order function that handles the application of its function argument f. We eliminate the modality of f before applying it to $()$ since f has type $[[\![\ell, E]\!]](1 \to [\![A]\!])$. We introduce a modality $\mathbf{mod}_{[[\![E]\!]]}$ for the whole translated function since $\textbf{handler}\ H$ is an effectful function with effect E. In the return clause of $[\![H]\!]$, we must eliminate the modality of x as shown in the typing rule T-Handle of Met($\mathcal{R}_{\text{scp}}$). This modality elimination is always possible as the type $[\![A]\!]$ of x always has kind Abs. The operation clause of $[\![H]\!]$ demonstrates why we must use modality-parameterised handlers. Note that rule T-Handler of System F$^\epsilon$ gives the continuation r in H the type $B' \to^E A$. By annotating **handle** with $[[\![E]\!]]$, the continuation r in $[\![H]\!]$ has type $[[\![E]\!]]([\![B']\!] \to [\![A]\!])$, which is equal to $[\![B' \to^E A]\!]$. We now give the full translation of the handler $sum_{\mathsf{F}^\epsilon}$ of Section 2.5.2.

$$
\begin{array}{rcl}
[\![sum_{\mathsf{F}^\epsilon}]\!] & = & \Lambda\varepsilon.\mathbf{mod}_{[\varepsilon]}\ (\lambda f^{[\texttt{yield},\varepsilon](1\to\texttt{Int})}.\textbf{handle}^{[\varepsilon]}\ (\textbf{let}\ \mathbf{mod}_{[\texttt{yield},\varepsilon]}\ f' = f\ \textbf{in}\ f\ ())\ \textbf{with} \\
 & & \{\textbf{return}\ x \mapsto \textbf{let}\ \mathbf{mod}_{[\texttt{yield},\varepsilon]}\ x' = x\ \textbf{in}\ x', \ell\ p\ r \mapsto p + (\textbf{let}\ \mathbf{mod}_{[\varepsilon]}\ r' = r\ \textbf{in}\ r'\ ())\}) \\
 & : & \forall\varepsilon.[\varepsilon]([\texttt{yield},\varepsilon](1 \to \texttt{Int}) \to \texttt{Int})
\end{array}
$$

We have the following type and semantics preservation theorems with proofs in **??**.

Theorem 4.1 (Type Preservation). *If $\Gamma \vdash M : A \mid E$ in System F$^\epsilon$, then $[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]\ @\ [\![E]\!]$ in Met($\mathcal{R}_{scp}$). Similarly for typing judgements of values.*

$$
\begin{array}{lll}
\text{Value Types} & A, B ::= 1 \mid T \ \mathbf{at}\ C & \text{Blocks} \quad P, Q ::= f \mid \{(\overline{x : A}, \overline{f : T}) \Rightarrow M\} \\
\text{Block Types} & T ::= (\overline{A}, \overline{f : T}) \Rightarrow B & \quad \mid \mathbf{unbox}\ V \\
\text{Capability Sets} & C ::= \{\overline{f}\} & \text{Computations} \quad M, N ::= \mathbf{return}\ V \mid P(\overline{V}, \overline{Q}) \\
\text{Contexts} & \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, f :^{C} T \mid \Gamma, f :^{*} T & \quad \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \\
\text{Values} & V, W ::= x \mid () \mid \mathbf{box}\ P & \quad \mid \mathbf{def}\ f = P\ \mathbf{in}\ N \\
\text{Handlers} & H ::= \{p\ r \mapsto N\} & \quad \mid \mathbf{try}\ \{f^{(A) \Rightarrow B} \Rightarrow M\}\ \mathbf{with}\ H
\end{array}
$$

$\boxed{\Gamma \vdash V : A} \quad \boxed{\Gamma \vdash P : T \mid C}$

$$
\text{T-Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \text{T-Var}\ \frac{\Gamma \ni x : A}{\Gamma \vdash x : A} \qquad \text{T-Box}\ \frac{\Gamma \vdash P : T \mid C}{\Gamma \vdash \mathbf{box}\ P : T\ \mathbf{at}\ C}
$$

$$
\text{T-Transparent}\ \frac{\Gamma \ni f :^{C} T}{\Gamma \vdash f : T \mid C} \qquad \text{T-Tracked}\ \frac{\Gamma \ni f :^{*} T}{\Gamma \vdash f : T \mid \{f\}}
$$

$$
\text{T-Unbox}\ \frac{\Gamma \vdash V : T\ \mathbf{at}\ C}{\Gamma \vdash \mathbf{unbox}\ V : T \mid C} \qquad \text{T-Block}\ \frac{\Gamma, \overline{x : A}, \overline{f :^{*} T} \vdash M : B \mid C \cup \{\overline{f}\}}{\Gamma \vdash \{(\overline{x : A}, \overline{f : T}) \Rightarrow M\} : (\overline{A}, \overline{f : T}) \Rightarrow B \mid C}
$$

$$
\text{T-BSub}\ \frac{\Gamma \vdash P : T \mid C' \qquad C' \subseteq C}{\Gamma \vdash P : T \mid C}
$$

$\boxed{\Gamma \vdash M : A \mid C}$

$$
\text{T-Value}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\ V : A \mid \cdot} \qquad \text{T-Let}\ \frac{\Gamma \vdash M : A \mid C \qquad \Gamma, x : A \vdash N : B \mid C'}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B \mid C \cup C'}
$$

$$
\text{T-Call}\ \frac{\Gamma \vdash P : (\overline{A_i}, \overline{f_j : T_j}) \Rightarrow B \mid C \qquad \overline{\Gamma \vdash V_i : A_i} \qquad \overline{\Gamma \vdash Q_j : T_j \mid C_j}}{\Gamma \vdash P(\overline{V_i}, \overline{Q_j}) : B[\overline{C_j / f_j}] \mid C \cup \overline{C_j}}
$$

$$
\text{T-Sub}\ \frac{\Gamma \vdash M : A \mid C' \qquad C' \subseteq C}{\Gamma \vdash M : A \mid C} \qquad \text{T-Def}\ \frac{\Gamma \vdash P : T \mid C' \qquad \Gamma, f :^{C'} T \vdash M : A \mid C}{\Gamma \vdash \mathbf{def}\ f = P\ \mathbf{in}\ M : A \mid C}
$$

$$
\text{T-Handle}\ \frac{\Gamma, f :^{*} (A') \Rightarrow B' \vdash M : A \mid C \cup \{f\} \qquad \Gamma, p : A', r :^{C} (B') \Rightarrow A \vdash N : A \mid C}{\Gamma \vdash \mathbf{try}\ \{f^{(A') \Rightarrow B'} \Rightarrow M\}\ \mathbf{with}\ \{p\ r \mapsto N\} : A \mid C}
$$

Fig. 4. Syntax and typing rules for System C. We mostly follow the syntax of Brachthäuser et al. [6]. The main difference is that we write $\Rightarrow$ for block types to emphasise they are second-class.

Theorem 4.2 (Semantics Preservation). *If M is well-typed and $M \rightsquigarrow N$ in System F^{ϵ}, then $[\![M]\!] \rightsquigarrow^{*} [\![N]\!]$ in $\textsc{Met}(\mathcal{R}_{scp})$ where $\rightsquigarrow^{*}$ denotes the transitive closure of $\rightsquigarrow$.*

5 Encoding a Capability-Based Effect System à la Effekt

In this section we briefly present System C [6], a core calculus formalising the capability-based effect system of Effekt [8], and show how to encode it into $\textsc{Met}(\mathcal{S})$. We refer to Brachthäuser et al. [6] for a complete introduction to System C.

5.1 System C

Figure 4 gives the syntax and typing rules for System C, which is fine-grain call-by-value [27] and distinguishes between first-class values V, blocks P (second-class functions), and computations M.

We have three typing judgements for values, blocks, and computations individually. Judgements for blocks $\Gamma \vdash P : T \mid C$ and computations $\Gamma \vdash M : A \mid C$ explicitly track a capability set C, which contains the capabilities in Γ that may be used.

The typing rules of System C are much more involved than those of System F^{ϵ} as capability tracking is deeply entangled with term constructs such as block constructions (T-Block), block calls (T-Call), block bindings (T-Def), and usages of block variables (T-Transparent and T-Tracked). Due to space constraints, we focus on explaining these key rules.

There are two rules for uses of block variables as there are two forms of block variable bindings in contexts. A *tracked* binding $f :^* T$ stands for a capability. Rule T-Tracked tracks f itself in the singleton capability set $\{f\}$. A *transparent* binding $f :^C T$ stands for a user-defined block whose capability set C is known. Rule T-Transparent tracks C as the capability set.

Rules T-Def and T-Block both bind block variables. Rule T-Def binds a block P as a transparent block variable $f :^{C'} T$ where C' is the capability set of P. Rule T-Block binds a list of tracked block variables (capabilities) $\overline{f :^* T}$ whose concrete capability sets are unknown until called. The rule T-Block reflects the roles that block constructions play for capability tracking as we introduced in Section 2.3.3. For instance, all capabilities $\overline{f}$ are added to the capability set of the block body M.

Rule T-Call fully applies a block P to values $\overline{V_i}$ and blocks $\overline{Q_j}$. The rule reflects the roles that block calls play for capability tracking as we introduced in Section 2.3.4. It substitutes each block variable f_j (recall that these variables are bound as $f_j :^* T$ in rule T-Block) with the capability set C_j of the block Q_j in type B. The capability set of the call is the union of the capability sets of P and all its block arguments because all these arguments might be invoked.

Rule T-Handle defines a named handler which introduces a capability $f : (A') \Rightarrow B'$ to the scope of M. Operation invocation via calling f in M is handled by this handler. The capability f is added to the capability set of M. The continuation r is introduced as a transparent binding with capability set C as it may only use capabilities in C provided by the context.

System C adopts named handlers and a generative semantics with a reduction relation $M \mid \Omega \rightsquigarrow N \mid \Omega'$ where $\Omega ::= \cdot \mid \ell : (A) \Rightarrow B$ is a context for runtime operation labels, similar to Met($\mathcal{X}$). The most interesting reduction rule is E-Gen which uses a runtime capability value $\mathbf{cap}_\ell$ with a runtime label ℓ to substitute a capability f introduced by a handler.

$$\text{E-Gen}\quad \mathbf{try}\ \{f^{(A)\Rightarrow B} \Rightarrow M\}\ \mathbf{with}\ H \mid \Omega \rightsquigarrow \mathbf{try}_\ell\ M[\mathbf{cap}_\ell/f]\ \mathbf{with}\ H \mid \Omega, \ell : (A) \Rightarrow B \quad \text{where } \ell \text{ fresh}$$

The full specification of operational semantics can be found in **??**.

5.2 Encoding System C in Met($\mathcal{S}$)

Figure 5 encodes System C in Met($\mathcal{S}$). The term translation is type-directed and defined on typing judgements. We annotate components of a term with their types and capability sets as necessary. We highlight syntax relevant to modalities and type abstraction of the term translation in grey. The grey parts show how modalities decouple capability tracking. The black parts remain valid programs after type erasure. The encoding is unavoidably more involved than that of System F^ϵ because of the deeper entanglement of capability tracking with blocks. As in Section 5.1, we focus on explaining the encoding of block-relevant constructs.

For block constructions and block calls, we have explained their encodings in detail in Sections 2.3.3 and 2.3.4, using the constructions and calls of blocks app_C and app'_C as examples.

A block binding **def** $f = P$ **in** N not only binds a block P to f but also annotate the binding $f :^{C'} T$ with the capability set C' of the block P as shown by rule T-Def in Figure 4. For instance, we can bind the block gen_C in Section 2.3.1 to f and apply it to 42. Its typing derivation is as follows.

$$\frac{y :^* \mathtt{Int} \Rightarrow \mathtt{1} \vdash gen_C \ : \ \mathtt{Int} \Rightarrow \mathtt{1} \mid \{y\} \qquad y :^* \mathtt{Int} \Rightarrow \mathtt{1}, f :^{\{y\}} \mathtt{Int} \Rightarrow \mathtt{1} \vdash f(42) \ : \ \mathtt{1} \mid \{y\}}{y :^* \mathtt{Int} \Rightarrow \mathtt{1} \vdash \mathbf{def}\ f = gen_C\ \mathbf{in}\ f(42) \ : \ \mathtt{1} \mid \{y\}}$$

The binding of f in the second premise is annotated with its capability set $\{y\}$ since gen_C uses the capability y. We cannot simply encode such a transparent binding by ignoring its annotation of the capability set. Instead, we use an absolute modality to simulate this annotation. To encode the binding of f, we wrap the translated block gen_C into the absolute modality $[y]$. The full translation of the above term is as follows, where we provide the omitted identity modality in Section 2.3.1.

$$\mathbf{let}\ f = \mathbf{mod}_{[y^*]}\ (\mathbf{mod}_{\langle\rangle}\ (\lambda x^{\mathtt{Int}}.\hat{y}\ x))\ \mathbf{in}\ \mathbf{let}\ \mathbf{mod}_{[y^*]}\ \hat{f} = f\ \mathbf{in}\ \mathbf{let}\ \mathbf{mod}_{\langle\rangle}\ f' = \hat{f}\ \mathbf{in}\ f'\ 42$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Cap Set} \to \text{Effect Context} \\
[\![\{\overline{f}\}]\!] & = & \overline{f^*}
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Value / Block Type} \to \text{Type} \\
[\![\mathtt{1}]\!] & = & \mathtt{1} \\
[\![T \ \mathbf{at}\ C]\!] & = & [[\![C]\!]][\![T]\!] \\
[\![(\overline{A}, \overline{f : T}) \Rightarrow B]\!] & = & \forall \overline{f^*}.\langle \overline{f^*} \rangle (\overline{[\![A]\!]} \to \overline{[f^*][\![T]\!]} \to [\![B]\!])
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Value} \to \text{Term} \\
[\![()]\!] & = & () \\
[\![x]\!] & = & x \\
[\![\mathbf{box}\ P : T \ \mathbf{at}\ C]\!] & = & \mathbf{mod}_{[[\![C]\!]]}\ [\![P]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Context} \to \text{Context} \\
[\![\cdot]\!] & = & \cdot \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], x : [\![A]\!] \\
[\![\Gamma, f :^* T]\!] & = & [\![\Gamma]\!], f^*, f : [f^*][\![T]\!], \hat{f} :_{[f^*]} [\![T]\!] \\
[\![\Gamma, f :^C T]\!] & = & [\![\Gamma]\!], f : [[\![C]\!]][\![T]\!], \hat{f} :_{[[\![C]\!]]} [\![T]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Block} \to \text{Term} \\
[\![f]\!] & = & \hat{f} \\
[\![\{(\overline{x : A}, \overline{f : T}) \Rightarrow M\}]\!] & = & \Lambda \overline{f^*}.\mathbf{mod}_{\langle \overline{f^*} \rangle}\ (\lambda \overline{x^{[\![A]\!]}}\ \overline{f^{[f^*][\![T]\!]}}. \\
 & & \qquad \overline{\mathbf{let}\ \mathbf{mod}_{[f^*]}\ \hat{f} = f\ \mathbf{in}}\ [\![M]\!]) \\
[\![\mathbf{unbox}\ V : T \mid C]\!] & = & \mathbf{let}\ \mathbf{mod}_{[[\![C]\!]]}\ x = [\![V]\!]\ \mathbf{in}\ x
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!] & : & \text{Computation / Handler} \to \text{Term} \\
[\![\mathbf{return}\ V]\!] & = & [\![V]\!] \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] & = & \mathbf{let}\ x = [\![M]\!]\ \mathbf{in}\ [\![N]\!] \\
[\![\mathbf{def}\ f = P : T \mid C\ \mathbf{in}\ N]\!] & = & \mathbf{let}\ f = \mathbf{mod}_{[[\![C]\!]]}\ [\![P]\!]\ \mathbf{in}\ \mathbf{let}\ \mathbf{mod}_{[[\![C]\!]]}\ \hat{f} = f\ \mathbf{in}\ [\![N]\!] \\
[\![P(\overline{V_i}, \overline{Q_j : T_j \mid C_j})]\!] & = & \mathbf{let}\ \mathbf{mod}_{\langle \overline{[\![C_j]\!]} \rangle}\ x = [\![P]\!]\ \overline{[\![C_j]\!]}\ \mathbf{in}\ x\ \overline{[\![V_i]\!]}\ \overline{(\mathbf{mod}_{[[\![C_j]\!]]}\ [\![Q_j]\!])} \\
\left[\!\!\left[\begin{array}{l}\mathbf{try}\ \{f^{(A')\Rightarrow B'} \Rightarrow M\} \\ \mathbf{with}\ H : A \mid C\end{array}\right]\!\!\right] & = & \mathbf{local}\ \ell_f : [\![A']\!] \twoheadrightarrow [\![B']\!]\ \mathbf{in}\ \mathbf{let}\ \mathbf{mod}_{\langle \ell_f \rangle}\ g = \\
 & & \qquad (\Lambda f^*.\mathbf{mod}_{\langle f^* \rangle}\ (\lambda f.\mathbf{let}\ \mathbf{mod}_{[f^*]}\ \hat{f} = f\ \mathbf{in}\ [\![M]\!]))\ \ell_f \\
 & & \mathbf{in}\ \mathbf{handle}^{[[\![C]\!]]}\ (g\ (\mathbf{mod}_{[\ell_f]}\ (\mathbf{mod}_{\langle\rangle}\ (\lambda x^{[\![A']\!]}.\mathbf{do}\ \ell_f\ x))))\ \mathbf{with}\ [\![H^{f,C}]\!] \\
[\![\{p\ r \mapsto N\}^{f,C}]\!] & = & \{\mathbf{return}\ x \mapsto \mathbf{let}\ \mathbf{mod}_{[\ell_f, [[\![C]\!]]]}\ x' = x\ \mathbf{in}\ x', \\
 & & \quad \ell_f\ p\ r \mapsto \mathbf{let}\ \mathbf{mod}_{[[\![C]\!]]}\ \hat{r} = r\ \mathbf{in}\ [\![N]\!]\}
\end{array}
$$

Fig. 5. An encoding of System C in Met($\mathcal{S}$).

We eliminate the modality $[y^*]$ of f and bind it to $\hat{f}$, reminiscent of how we translate block arguments bound by block constructions. In general, for a transparent block variable binding $f :^C T$ in the context, it is translated to two variable bindings $f : [[\![C]\!]][\![T]\!]$ and $\hat{f} :_{[[\![C]\!]]} [\![T]\!]$.

The translation of uses of block variables is simple. We translate each f to its hat version $\hat{f}$. The simplicity benefits from the fact that we eagerly eliminate the modality of each f after it is introduced, e.g., in the translations of block constructions and block bindings.

The translation of named handlers $\mathbf{try}\ \{f^{(A')\Rightarrow B'} \Rightarrow M\}\ \mathbf{with}\ H$ is different from the translation of sum_C in Section 2.5.3. The full translation of sum_C is as follows, where we provide the omitted identity modality of the function $\lambda x^{\mathtt{Int}}.\mathbf{do}\ \ell_y\ x$.

$$
\begin{array}{l}
\mathbf{local}\ \ell_y : \mathtt{Int} \twoheadrightarrow \mathtt{1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{mod}_{\langle \ell_y \rangle}\ g = (\Lambda y^*.\mathbf{mod}_{\langle y^* \rangle}\ (\lambda y.\mathbf{let}\ \mathbf{mod}_{[y^*]}\ \hat{y} = y\ \mathbf{in}\ \hat{y}\ 42; \hat{y}\ 37; 0))\ \ell_y \\
\mathbf{in}\ \mathbf{handle}^{[[\![C]\!]]}\ (g\ (\mathbf{mod}_{[\ell_y]}\ (\mathbf{mod}_{\langle\rangle}\ (\lambda x^{\mathtt{Int}}.\mathbf{do}\ \ell_y\ x)))) \\
\quad \mathbf{with}\ \{\mathbf{return}\ x \mapsto \mathbf{let}\ \mathbf{mod}_{[\ell_y, [[\![C]\!]]]}\ x' = x\ \mathbf{in}\ x', \ell_y\ p\ r \mapsto \mathbf{let}\ \mathbf{mod}_{[[\![C]\!]]}\ \hat{r} = r\ \mathbf{in}\ p + \hat{r}\ ()\}
\end{array}
$$

The main difference is that, instead of directly using the local label ℓ_y for the handled computation, we introduce an effect variable y^* first and substitute it with ℓ_y. This extra layer of abstraction is necessary to keep the translation systematic, because our translations of types and terms consistently translate a capability y to an effect variable y^*. After reducing the type application and substitution of g in the above translation term, we get the translation of sum_C in Section 2.5.3.

In the return clause, we additionally eliminate the modality of x. In the operation clause, we eliminate the modality $[[\![C]\!]]$ of r and bind it to $\hat{r}$ as we use a modality-parameterised handler. Using

a modality-parameterised handler is important because in sum_C, the continuation r is a transparent binding of form $f :^C \mathtt{1} \to \mathtt{Int}$ as shown by the typing rule T-Handle of System C in Section 5.1. We need to wrap the translated continuation r with the absolute modality $[[\![C]\!]]$ to be consistent with the translation of transparent bindings.

For contexts, we translate each entry. For a variable binding $x : A$, we translate it homomorphically. For a transparent binding of a block variable $f :^C T$, we translate it to two term variables f and $\hat{f}$ as discussed in the translation of **def** above. For a tracked binding of a block variable $f :^* T$, we translate it to an effect variable f^* and two term variables f and $\hat{f}$ as discussed in Section 2.2.

We have the following type and semantics preservation theorems with proofs in **??**.

Theorem 5.1 (Type Preservation). *If $\Gamma \vdash M : A \mid C$ in System C, then $[[\![\Gamma]\!]] \vdash [[\![M]\!]] : [[\![A]\!]] \; @ \; [[\![C]\!]]$ in Met($\mathcal{S}$). Similarly for typing judgements of values and blocks.*

Theorem 5.2 (Semantics Preservation). *If M is well-typed and $M \mid \Omega \rightsquigarrow N \mid \Omega'$ in System C, then $[[\![M]\!]] \mid [[\![\Omega]\!]] \rightsquigarrow^* [[\![N]\!]] \mid [[\![\Omega']\!]]$ in Met($\mathcal{S}$), where $\rightsquigarrow^*$ denotes the transitive closure of $\rightsquigarrow$.*

6 More Encodings and Discussions

In this section, we discuss more encodings of effect systems into Met($\mathcal{X}$), highlight practical language design insights gleaned from our encodings, and outline potential extensions to Met($\mathcal{X}$).

6.1 An Early Version of Effekt

System Ξ [7] is an early core calculus of the Effekt language. System Ξ is essentially a fragment of System C without boxes. As a result, in System Ξ capabilities can never appear in types since we cannot box a second-class block into a first-class value. While our encoding of System C in Section 5.2 directly gives an encoding of System Ξ in Met($\mathcal{S}$), it introduces unnecessary complexity. Since capabilities never appear in types in System Ξ, we do not need to introduce an effect variable f^* for each capability f in the encoding. It turns out that we can simply encode second-class blocks in System Ξ as first-class functions in Met($\mathcal{S}$) without introducing any extra term constructs. For instance, a block $\{(x : A, f : T) \Rightarrow M\}$ is encoded as a function $\lambda x^{[[\![A]\!]]} \, f^{[[\![T]\!]]}.[[\![M]\!]]$ by merely changing the notations. We provide the full encoding of System Ξ in Met($\mathcal{S}$) in **??** and prove it preserves types and semantics in **??**.

6.2 Named Handlers in Koka

Xie et al. [41] extend Koka with named handlers and formalise this extension in the core calculus System $\mathsf{F}^{\epsilon+\mathsf{sn}}$, which is based on System F^{ϵ}. System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ allows each handler to bind a handler name that can be used to invoke operations. A handler name is similar to a capability in System C but it is a first-class value. For instance, we can define a named handler in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ as follows.

$$sum_{\mathsf{F}^{\epsilon+\mathsf{sn}}} \doteq \Lambda\varepsilon.\textbf{nhandler} \; \{\mathtt{yield} \; p \; r \mapsto p + r \; ()\} \; : \forall\varepsilon.(\forall a.\mathsf{ev} \; \mathtt{yield}^a \to^{\mathtt{yield}^a,\varepsilon} \mathtt{Int}) \to^{\varepsilon} \mathtt{Int}$$

This handler is similar to the handler $sum_{\mathsf{F}^{\epsilon}}$ in Section 2.5.2. The main difference is that the argument takes a value of type $\mathsf{ev} \; \mathtt{yield}^a$. This is a first-class handler name with which we can invoke the `yield` operation. For example, we can apply $sum_{\mathsf{System\ F}^{\epsilon+\mathsf{sn}}}$ as follows.

$$sum_{\mathsf{F}^{\epsilon+\mathsf{sn}}} \; E \; (\Lambda a.\lambda h^{\mathsf{ev} \; \mathtt{yield}^a}.h \; 42; h \; 37; 0)$$

Instead of using the label `yield` to invoke the operation as in application of $sum_{\mathsf{F}^{\epsilon}}$ in Section 2.5.2, we directly apply the handler name h to arguments. This is reminiscent of the handler $sum_{\mathsf{System\ C}}$ in Section 2.5.3 where we invoke the operation by calling the capability introduced by the handler. This program reduces to 79. The scope variable a ensure scope safety of the handler name, similar to the technique used by `runST` in Haskell [22].

As with the encoding of named handlers in System C, we can encode a named handler of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ by introducing a local label ℓ_a and using the term $\mathbf{mod}_{[\ell_a]}\ (\lambda x.\mathbf{do}\ \ell_a\ x)$ to simulate the handler name. We use the effect structure $\mathcal{S}$ instead of $\mathcal{R}_{\mathsf{scp}}$ as there can never be duplicated handlers with the same name in System $\mathsf{F}^{\epsilon+\mathsf{sn}}$. The theory $\mathcal{S}$ gives us flexibility to have multiple effect variables, which we use to encode scope variables. We give the full encoding of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ in $\textsc{Met}(\mathcal{S})$ in **??** and prove its type and semantics preservation in **??**.

6.3 Insights for Language Design

In Section 2.4 and Section 2.5.4, we demonstrated how our encodings provide a direct way to compare the differences of System F^{ϵ} and System C. Moreover, our encodings can also help to inform language design choices based on the following observations.

(1) Our encodings together demonstrate that modal effect types are as expressive as the row-based and capability-based effect systems we consider.
(2) The encoding of System Ξ (Section 6.1) implies that we need not sacrifice first-class functions in order to obtain the benefits of the contextual effect polymorphism of Effekt.
(3) The encodings of System C (Section 5.2), System Ξ (Section 6.1), and System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ (Section 6.2) demonstrate that we can use local labels, a minimal extension as introduced in Section 3, to simulate the relatively heavyweight feature of named handlers in Effekt and Koka.
(4) The encoding of System $\mathsf{F}^{\epsilon+\mathsf{sn}}$ (Section 6.2) further demonstrates that the first-class handler names of Koka offer no extra expressiveness over the second-class local labels of $\textsc{Met}(\mathcal{X})$.
(5) The encoding of System C (Section 5.2) shows that instead of having a built-in form of capabilities which can appear at both term and type levels as in Effekt and Scala [5], 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.

6.4 Potential Extensions to $\textsc{Met}(\mathcal{X})$

We discuss three potential extensions to $\textsc{Met}(\mathcal{X})$ and leave their full development as future work.

Effect Kinds. We can extend the effect structure to abstract over effect kinds instead of having a single kind Effect. The augmented definition of effect structure is a triple $\mathcal{X} = \langle \mathbb{R}, :, \equiv \rangle$ where the new component $\mathbb{R}$ is a set of effect kinds. We must extend the kinding and equivalence relations accordingly. As an example of this extension, in order to characterise Rémy-style row types [35] which use a kind system to ensure that there is no duplicated label, we can declare $\mathbb{R} = \{\mathsf{Row}_{\mathcal{L}} \mid \mathcal{L}\}$ where $\mathcal{L}$ is a label set and denotes all labels that must not be in the row. As another example, this extension enables us to combine different effect structures together by assigning a kind to each theory. For instance, we can declare two kinds Set and Row for theories $\mathcal{S}$ and $\mathcal{R}_{\mathsf{scp}}$ respectively, and then give local labels the kind Set and global labels the kind Row. We can then treat local labels as sets and global labels as scoped rows.

Presence Types. We can associate operation labels in extensions and effect contexts with presence types [36]. Furthermore, instead of predefining the operation types for labels, we can assign operation types to labels in extensions and effect contexts in the manner of Tang et al. [38]. For instance, the syntax of extensions could be extended to $D ::= \cdot \mid \ell : P, D \mid \varepsilon, D$, where P is a presence type typically defined as $P ::= - \mid \mathsf{Pre}(A \twoheadrightarrow B) \mid \theta$. A label can be absent ($-$), present with a type ($\mathsf{Pre}(A \twoheadrightarrow B)$), or polymorphic over its presence (θ).

Masking. $\textsc{Met}(\mathcal{X})$ does not include the mask operator and the mask modality $\langle L \rangle$ of Met [38]. This enables us to substantially simplify the presentation of the core calculus, especially the definitions relevant to modalities in Section 3.3, compared to that of Tang et al. [38]. Moreover, the

lack of the mask operator does not influence our encodings as the core calculi of Effekt and Koka do not have it. Masking [2, 10] is useful for effect systems based on scoped rows where duplicated labels indicate nested handlers for the same operation label. With the mask operator, we can manually select which handler to use when nested. It is interesting future work to extend $\textsc{Met}(\mathcal{X})$ with a suitable notion of abstract mask operator and extend the syntax of relative modalities to $\langle L | D \rangle$ where L is a mask and D is an extension. This extension will require extending the effect structure to define the kinding and equivalence relations of masks. A form of masking also makes sense for effect structures other than $\mathcal{R}_{\text{scp}}$. For instance, masking ℓ from a computation in $\mathcal{S}$ could be used to disallow ℓ to be performed by the computation.

7 Related and Future Work

Row-Based Effect Systems. Row-based effect systems track effects by annotating function arrows with row types denoting effects. They have been adopted in research languages such as Links [16], Koka [24], and Frank [28]. Links uses Rémy-style row types with presence polymorphism [36], whereas Koka and Frank use scoped rows [23]. Eff [1] and Helium [4] also track effects on function arrows but treat effect types as sets. In this paper we focus on Koka, but we expect that other row-based effect systems can be encoded similarly by instantiating the effect structure appropriately.

Capability-Based Effect Systems. Capability-based effect systems introduce and track effects as capabilities. Different variations diverge on when capability sets appear in types. Effekt [6, 7] uses second-class functions and only attaches capability sets to types when boxing functions. $\text{CC}_{<:\Box}$ [5] and Capless [42], the foundations for capture tracking in Scala 3, always annotate every type with its capability set and use subtyping and syntactic sugar to simplify capability sets. It is interesting future work to encode them in $\textsc{Met}(\mathcal{X})$.

Abstracting Effect Systems. Yoshioka et al. [43] study different treatments of effect collections in row-based effect systems. They propose a parameterised core calculus, λ_{EA}, whose effect types can be instantiated to various kinds of sets and rows. The effect types in λ_{EA} are still entangled with function types. As a result, λ_{EA} cannot encode capability-based effect systems. We follow λ_{EA} in parameterising our core calculus $\textsc{Met}(\mathcal{X})$ over different treatments of effect collections. We make use of modalities to decouple effect tracking from function types, enabling the encodings of both row-based and capability-based effect systems.

Encoding into Modal Effect Types. Tang et al. [38] consider a restricted row-based effect system in which each effect type can refer only to the lexically closest effect variable. This restricted system remains remarkably expressive and suffices for many practical programs. Nonetheless, they show that it can be encoded into simply-typed $\textsc{Met}$ without any effect polymorphism. Our encodings consider richer source languages, showing that modal effect types are as expressive as several row-based and capability-based effect systems in the literature.

Local Effects. Local labels in $\textsc{Met}(\mathcal{X})$ allow us to introduce fresh effects locally. They are useful for solving the effect encapsulation and accidental handling problems [3, 10, 44]. As discussed in Section 2.5.3, there are various local effect formalisms in the literature [3, 11, 19]; most are based on dynamic generation of fresh effect names, whereas the calculus of Biernacki et al. [3] is based on effect coercions. We conjecture that local labels of $\textsc{Met}(\mathcal{X})$ are as expressive as these formalisms. We are interested in studying their relationship by encoding them into $\textsc{Met}(\mathcal{X})$.

A Modal Type System for Benign Effects. Nanevski [31] propose a modal type system for benign effects in Chapter 4.3. We refer to this system as MTBE. MTBE supports local effects and indexes the standard necessity modality $\Box$ with effects for effect tracking. In MTBE, a type $\Box_E\, A$ means a

computation which returns a value of type A and may perform effects in E. This indexed necessity modality is similar to the absolute modality of $\textsc{Met}(\mathcal{X})$. The key difference between MTBE and $\textsc{Met}(\mathcal{X})$ (and Met) is that MTBE has no notion of ambient effect context. MTBE requires functions to be pure: every function type must specify all the effects it may perform via a box. In contrast, $\textsc{Met}(\mathcal{X})$ allows a function to perform any effects from the ambient effect context. For example, an application function of type $(\texttt{Int} \longrightarrow \texttt{1}) \longrightarrow \texttt{Int} \longrightarrow \texttt{1}$ in $\textsc{Met}(\mathcal{X})$ allows its argument to perform any effects from the ambient effect context, whereas a function with such a type in MTBE can only be applied to pure functions (by default each function type has the empty $\Box$). In order to apply an application function to effectful arguments in MTBE we must specify what effects may be performed in the type. This requires parametric effect polymorphism in order to support arbitrary effectful arguments. Moreover, MTBE does not support relative modalities as relative modalities are intimately tied to the notion of ambient effect contexts. Our encoding of System C in Section 5.2 relies on the notion of ambient effect contexts and relative modalities. As a result, MTBE cannot serve as a general framework for encoding various effect systems as $\textsc{Met}(\mathcal{X})$ does.

Effectful Contextual Modal Type Theory. Zyuzin and Nanevski [45] propose effectful contextual modal type theory (ECMTT) which extends the *contextual necessity modality* [32] to track contexts of effectful operations. Similar to MTBE, ECMTT also lacks the notion of ambient effect contexts and is thus less expressive and flexible than $\textsc{Met}(\mathcal{X})$. Moreover, ECMTT does not support dynamic generation of fresh effect names and thus cannot express named handlers as in Effekt.

Call-By-Push-Value. Attempts to decouple programming language features have frequently born fruit. For instance, call-by-push-value (CBPV) [26] subsumes both call-by-value (CBV) and call-by-name (CBN) by decoupling thunking and forcing from function abstraction and application. Our work is in a similar vein. More interestingly, our encodings of System F^{ϵ} and System C possess certain similarities with Levy's encodings of CBV and CBN into CBPV, respectively. In our encoding of System F^{ϵ}, each function is wrapped in an absolute modality, reminiscent of the CBV-to-CBPV encoding where each function is thunked. In our encoding of System C, we only wrap a block in an absolute modality when passing it as an argument, reminiscent of the CBN-to-CBPV encoding, in which thunking of a function is deferred until passing it as an argument. We are interested in further exploring these similarities.

Expressive Power of Effect Handlers. Forster et al. [13] compare the expressive power of effect handlers, monadic reflection, and delimited control in a simply-typed setting and show that delimited control cannot encode effect handlers in a type-preserving way. Piróg et al. [33] extend the comparison between effect handlers and delimited control to a polymorphic setting and show their equivalence. Ikemori et al. [18] further show the typed equivalence between named handlers and multi-prompt delimited control. In contrast to these works, which compare effect handlers with other programming abstractions, we compare different effect systems for effect handlers.

Future Work. In addition to the ideas already discussed above and in Section 6.4, other directions for future work include: exploring inverse encodings (from instantiations of $\textsc{Met}(\mathcal{X})$ into other calculi); studying parametricity and abstraction safety [4, 44] for $\textsc{Met}(\mathcal{X})$; and further developing $\textsc{Met}(\mathcal{X})$ as a uniform intermediate language for type- and effect-directed optimisation.

Acknowledgments

We thank Jonathan Immanuel Brachthäuser, Anton Lorenzen, Orpheas van Rooij, Jesse Sigal, and the anonymous reviewers of ICFP 2025 and POPL 2026 for feedback. Sam Lindley was supported by UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (MR/T043830/1 and MR/Z000351/1).

References

[1] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Log. Methods Comput. Sci.* 10, 4 (2014). doi:10.2168/LMCS-10(4:9)2014

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* 2, POPL (2018), 8:1–8:30. doi:10.1145/3158096

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. doi:10.1145/3290319

[4] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. doi:10.1145/3371116

[5] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. doi:10.1145/3618003

[6] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. doi:10.1145/3527320

[7] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. doi:10.1145/3428194

[8] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2025. Effekt Language: A language with lexical effect handlers and lightweight effect polymorphism. https://effekt-lang.org. Accessed 2025-07-10.

[9] Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering purity with comonads and capabilities. *Proc. ACM Program. Lang.* 4, ICFP (2020), 111:1–111:28. doi:10.1145/3408993

[10] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9. doi:10.1017/S0956796820000039

[11] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 225–252. doi:10.1007/978-3-031-30044-8_9

[12] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. doi:10.1016/0167-6423(91)90036-W

[13] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15. doi:10.1017/S0956796819000121

[14] Daniel Gratzer. 2023. *Syntax and semantics of modal type theory.* Ph. D. Dissertation. Aarhus University.

[15] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/LMCS-17(3:11)2021

[16] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/2976022.2976033

[17] Alex Hubers and J. Garrett Morris. 2023. Generic Programming with Extensible Data Types: Or, Making Ad Hoc Extensible Data Types Less Ad Hoc. *Proc. ACM Program. Lang.* 7, ICFP (2023), 356–384. doi:10.1145/3607843

[18] Kazuki Ikemori, Youyou Cong, and Hidehiko Masuhara. 2023. Typed Equivalence of Labeled Effect Handlers and Labeled Delimited Control Operators. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, Santiago Escobar and Vasco T. Vasconcelos (Eds.). ACM, 4:1–4:13. doi:10.1145/3610612.3610616

[19] Robin Jourde. 2022. M1 Internship Report : Effect Typing for Links. https://github.com/Orbion-J/intership-report-2022/blob/master/pdf/report.pdf Accessed 2025-07-10.

[20] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. doi:10.1145/2500365.2500590

[21] G. A. Kavvos and Daniel Gratzer. 2023. Under Lock and Key: a Proof System for a Multimodal Logic. *Bull. Symb. Log.* 29, 2 (2023), 264–293. doi:10.1017/BSL.2023.14

[22] John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP Symb. Comput.* 8, 4 (1995), 293–341.

[23] Daan Leijen. 2005. Extensible records with scoped labels. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005 (Trends in Functional Programming, Vol. 6)*, Marko C. J. D. van Eekelen (Ed.). Intellect, 179–194.

[24] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. doi:10.1145/3009837.3009872

[25] Daan Leijen. 2025. Koka: A strongly typed functional-style language with effect types and handlers. https://koka-lang.github.io. Accessed 2025-07-10.

[26] Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.
[27] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9
[28] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 500–514. doi:10.1145/3009837.3009897
[29] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (2024), 485–514. doi:10.1145/3674642
[30] J. Garrett Morris and James McKinna. 2019. Abstracting Extensible Data Types: Or, Rows by Any Other Name. *Proc. ACM Program. Lang.* 3, POPL, Article 12 (jan 2019), 28 pages. doi:10.1145/3290325
[31] Aleksandar Nanevski. 2004. *Functional programming with names and necessity.* Ph. D. Dissertation. USA. AAI3143944.
[32] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. doi:10.1145/1352582.1352591
[33] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16. doi:10.4230/LIPICS.FSCD.2019.30
[34] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013
[35] D. Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/75277.75284
[36] Didier Rémy. 1994. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Citeseer.
[37] Wenhao Tang and Sam Lindley. 2025. Rows and Capabilities as Modal Effects. arXiv:2507.10301 [cs.PL] https://arxiv.org/abs/2507.10301
[38] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1130–1157. doi:10.1145/3720476
[39] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.
[40] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP (2020), 99:1–99:29. doi:10.1145/3408981
[41] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 30–59. doi:10.1145/3563289
[42] Yichen Xu, Oliver Bračevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the Box: Ergonomic and Expressive Capture Tracking over Generic Data Structures. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 334 (Oct. 2025), 28 pages. doi:10.1145/3763112
[43] Takuma Yoshioka, Taro Sekiyama, and Atsushi Igarashi. 2024. Abstracting Effect Systems for Algebraic Effect Handlers. *CoRR* abs/2404.16381 (2024). doi:10.48550/ARXIV.2404.16381 arXiv:2404.16381
[44] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. doi:10.1145/3290318
[45] Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual modal types for algebraic effects and handlers. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473580

Received 2025-07-10; accepted 2025-11-06