Modal Effect Types

WENHAO TANG, The University of Edinburgh, United Kingdom
LEO WHITE, Jane Street, United Kingdom
STEPHEN DOLAN, Jane Street, United Kingdom
DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom
ANTON LORENZEN, The University of Edinburgh, United Kingdom

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

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

Additional Key Words and Phrases: effect handlers, effect types, modal types, multimodal type theory

ACM Reference Format:
Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 120 (April 2025), 28 pages. https://doi.org/10.1145/3720476

1 Introduction

Effect handlers [43] allow programmers to define, customise, and compose a range of computational effects including concurrency, exceptions, state, backtracking, and probability, in direct-style inside the programming language. Following their pioneering use in languages such as Eff [4], Effekt [8, 9], Frank [13, 33], Koka [32], and Links [21], they are now increasingly being adopted in production languages and systems such as OCaml [49], Scala [7], and WebAssembly [41].

In a statically typed programming language with effect handlers some form of effect system to track effectful operations is necessary in order to ensure that a given program handles all of its effects. However, traditional effect systems require extensive effect annotations even for code that does not use effects. Consider the standard `map` function:

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

This type is a statement about the values that map accepts and returns, but is silent about which effects may occur during its evaluation. In the effect system of Koka, for instance, this `map` function is thus presumed to be a total function that takes a function which cannot perform any effects and itself does not perform any effects.

Authors' Contact Information: Wenhao Tang, wenhao.tang@ed.ac.uk, The University of Edinburgh, United Kingdom; Leo White, lwhite@janestreet.com, Jane Street, United Kingdom; Stephen Dolan, sdolan@janestreet.com, Jane Street, United Kingdom; Daniel Hillerström, daniel.hillerstrom@ed.ac.uk, The University of Edinburgh, United Kingdom; Sam Lindley, sam.lindley@ed.ac.uk, The University of Edinburgh, United Kingdom; Anton Lorenzen, anton.lorenzen@ed.ac.uk, The University of Edinburgh, United Kingdom.

ACM 2475-1421/2025/4-ART120
https://doi.org/10.1145/3720476

However, this would prevent programmers from passing any effectful function to `map`. To use `map` in effectful code, in Koka we must give it a more permissive type such as:

```
map' : ∀ a b e . (a -e-> b) -e-> List a -e-> List b
```

This type uses *effect polymorphism*, quantifying over an *effect variable* `e`, which occurs on every arrow. Such effect annotations pollute the type signature of `map` to convey the obvious: `map'` is polymorphic in its effects, that is, the effects of `map' f xs` depend on the effects of the function argument `f`. Effect annotations impose a mild burden to authors of new code, but pose a significant problem when extending an existing language with effectful features.

Type signatures of existing library code must be rewritten to support effect polymorphism [7, 38], even in legacy libraries that do not use effects, making it challenging to retrofit such an effect system onto an existing language in a backwards-compatible way without causing friction for existing codebases. However, if we can eliminate the need to annotate effect polymorphism, then retrofitting an effect system ought to become a tractable problem. Our goal is to design a principled effect system, where effect polymorphism silence is a virtue.

An important step towards that goal was taken by the language Frank [13, 33]. Frank gives `map` its original unannotated type, whilst still allowing it to be passed effectful functions. The key idea is that expressions are typed assuming an unknown set of possible effects—the *ambient effects*—will be provided by the context in which the expression occurs. Rather than assuming unannotated function types perform no effects, they are assumed to perform the ambient effects.

Frank still uses effect variables behind the scenes, implicitly inserting effect variables for passing the ambient effects around. For instance, Frank simply treats the type signature of `map` above as syntactic sugar for `map'` (Frank has certain other syntactic idiosyncrasies, so in order to ease readability, we render Frank code in similar syntax to Metl, which we introduce in Section 2). This syntactic mechanism is fragile. For instance, effect variables can appear in error messages as in the following example in which we use a `yield` effect to write a function that yields all values in a list.

```
gen : List Int -yield-> 1
gen xs = map (fun x → do yield x) xs; ()
```

If the user forgets the `yield` annotation, Frank complains:

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

Here `£` and `e` are the underlying effect variables inserted by the Frank compiler. They do not appear in the source program and in larger programs it can be unclear how to fix such errors.

Effekt [9] and Scala [7] also make use of ambient effects to avoid effect polymorphism by tracking effects as capabilities. However, they either restrict functions to be second-class or require having capability variables in types for certain use cases, as we discuss further in Section 8.1.

We build on the insight that ambient effect contexts can substantially reduce the annotation burden. Instead of relying on desugaring to traditional effect polymorphism like Frank, we develop Met (Modal Effect Types), a novel effect system with a theoretical foundation based on modal types. We follow multimodal type theory (MTT) [18, 19] in tracking *modes* for types and terms and consider *modalities* as the transitions between such modes. We treat each possible ambient effect context as a mode, and each possible transition between effect contexts as a modality. Our system provides two kinds of modalities: 1) *absolute* modalities, which override the ambient effect context; and 2) *relative* modalities, which describe a local change to the ambient effect context, as exemplified by effect handlers which handles certain effects and forwards others unchanged.

Met precludes hidden effect variables in error messages as there are no hidden effect variables. Moreover, Met works smoothly with pure first-class higher-order functions, which require neither hidden effect variables nor extra annotations, and can be applied to effectful arguments. Both Frank

and Met strive to capture the essence of modular programming with effects. Frank relies on a fragile syntactic characterisation based on polymorphic types. In contrast, Met provides a more robust characterisation based on simple types and modal types.

The main contributions of this paper are as follows.

- We give a high-level overview of the key ideas of modal effect types: effect contexts, absolute and relative modalities. We provide a series of practical examples to show how modal effect types enable us to write modular effectful programs without effect polymorphism (Section 2).
- We briefly recall the design of multimodal type theory (MTT), the basis of modal effect types, and outline why MTT works well for designing an effect system (Section 3).
- We introduce Met, a simply-typed core calculus with effect handlers and modal effect types (Section 4). We prove its type soundness and effect safety.
- Intuitively, Met can type check all functions that can be written in traditional effect systems using a single effect variable, which is the most common case in practice. We formally prove this intuition by presenting a calculus for row-based effect systems with a single effect variable and encoding it in Met (Section 5).
- We extend Met with data types and polymorphism for value types. To recover the full power of traditional effect systems, we also extend Met with effect polymorphism which can be seamlessly used alongside modal effect types to express effectful programs that use higher-order effects modularly (Section 6).
- We outline and prototype a surface language Metl which uses bidirectional type checking to infer the introduction and elimination of modalities (Section 7).
- We present a direct comparison of type signatures in Metl, Koka, and Effekt in order to highlight the practical appeal of modal effect types (Section 8.2).

Section 8 also discusses related and future work. The full specifications, proofs, and appendices can be found in the extended version of the paper [51].

2 Programming with Modal Effect Types

In this section we illustrate the main ideas of modal effect types through a series of examples. We demonstrate how modal effect types support modular composition of higher-order functions and effect handlers without effect polymorphism. The examples are written in Metl, which translates to the core calculus Met via a simple type-directed elaboration. In order to elucidate the core idea that modal effect types support modular effectful programming without polymorphism we begin with examples in the simply-typed fragment of Metl.

2.1 From Function Arrows to Effect Contexts

Traditional effect systems annotate a function type with the effects that the function may perform when invoked. For instance, consider the following typing judgement for the `app` function specialised to take a pair of a function from integers to the unit type and an integer.

```
⊢ fun (f, x) → f x : (Int -E-> 1, Int) -E-> 1
```

The effect annotation `E` is a row of typed operations that `f` may perform. For instance, if `E` is `get : 1 ↠ Int, put : Int ↠ 1` then `f` may perform a `get` operation which takes a unit value and returns an integer and a `put` operation which takes an integer and returns a unit value. As in Frank [13] and Koka [32], rows are *scoped* [31] meaning that they allow duplicate operations with the same name (but possibly different types). The order of duplicates matters, but the relative order of distinct operations does not.

Since `app` invokes its argument, `E` also denotes the operations that invoking `app` might perform. As we saw in the introduction, the standard way to support modularity is to be polymorphic in `E`. But this introduces an annotation burden for all higher-order functions, including those (like `app`) which do not themselves perform effects.

In the spirit of Frank, Met decouples effects from function types and tracks *effect contexts* in typing judgements. All components of the term and type share the same effect context (unless manipulated by modalities as we will see in Sections 2.2 and 2.3). For instance, we have the following typing judgement for the same `app` function as above.

```
⊢ fun (f, x) → f x : (Int → 1, Int) → 1 @ E
      @E     @E      @E             @E
```

As a visual aid, we use braces to explicitly annotate the effect contexts for the argument `f` and the whole function in the term and type. The `@ E` annotation belongs to the judgement and indicates the effect context `E`. This is the *ambient effect context* for the whole term and type of this typing judgement. An effect context specifies which operations may be performed. In this example, the effect contexts are all the same as the ambient effect context. We know that `app` can perform the same effects as its argument `f` as they share the same effect context.

2.2 Overriding the Ambient Effect Context with Absolute Modalities

An *absolute modality* `[E]` defines a new effect context `E` that overrides the ambient effect context. For instance, the following function invokes the operation `yield` via the **`do`** keyword. The `yield` operation takes an integer and returns a unit value.

```
⊢ fun x → do yield x : [yield : Int ↠ 1](Int → 1) @ .
          @ yield : Int ↠ 1          @ yield : Int ↠ 1
```

The absolute modality `[yield : Int ↠ 1]` specifies a singleton effect context in which the `yield` operation with type `Int ↠ 1` may be performed. Here, it overrides the empty ambient effect context (`.`), allowing `yield` to be performed in the function body.

Effect contexts percolate through the structure of a type. For example, a function of type `[E](A → B)` may perform effects `E` when invoked, and a list of type `[E](List (A → B))` may perform effects `E` when its components are invoked. For brevity, we define an effect context abbreviation.

```
eff Gen a = yield : a ↠ 1
```

Such abbreviations are merely macros, such that, for instance, `[Gen Int]` denotes the modality `[yield : Int ↠ 1]` and `[Gen Int, E]` denotes the modality `[yield : Int ↠ 1, E]`.

For higher-order functions like `map` and `app` which do not directly perform any effects, we use the empty absolute modality `[]`. For instance, in Metl, the curried first-class higher-order `iter` function, specialised to iterate over a list of integers, is defined as follows.

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

The empty absolute modality `[]` specifies an empty effect context in which the function is defined. However, due to subeffecting, `iter` is not limited to only the empty effect context. For instance, we can apply `iter` to the previous function which uses `yield`.

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

Metl allows us to use `iter` here directly even though its type contains an absolute modality. This is allowed since an elaboration step implicitly eliminates the modality of `iter` before it is applied to the function that invokes `yield`. Following the literature on modal types, we refer to introduction

of modalities as *boxing* and elimination as *unboxing*. Though the modality `[]` requires us to use `iter` under the empty effect context, subeffecting then upcasts it to the singleton one `Gen Int`.

To achieve the same flexibility of applying `iter` to any effectful arguments in a traditional row-based effect system, we would need effect polymorphism:

```
iter : ∀ e . (Int -e-> 1) -e-> List Int -e-> 1
```

2.3 Transforming the Ambient Effect Context with Relative Modalities

So far we have only seen examples that are either pure or just perform effects. Absolute modalities suffice for modular programming with such examples without requiring any use of effect polymorphism. However, the situation becomes more interesting when we introduce constructs that manipulate effect contexts non-trivially, such as effect handlers. Effect handlers provide a way of interpreting effects inside the language itself. For instance, we can use an effect handler to interpret operations `yield : Int ↠ 1` by collecting their arguments into a list of integers.

```
asList f = handle f () with
  return ()              ↦ nil
  (yield : Int ↠ 1) x r ↦ cons x (r ())
```

The body of `asList` invokes the function `f` inside a *handler*. The handler has two clauses that account for two cases: 1) what happens when `f` returns; and 2) what happens when `f` performs `yield`. In the first case, it returns the list `nil`. In the second case, it prepends the integer `x` onto the head of the list returned by the application of `r`. Here `r` is bound to the continuation of the `yield` invocation inside `f`. The argument type of `r` is determined by the return type of the operation being handled (unit in the case of `yield`) and its return type is determined by the return type of the handler. Thus `r : 1 → List Int`. When the continuation `r` is invoked, the handler is reinstalled around it to handle the residual effects of `f` (this kind of handler is known as *deep* in the literature [27]). We write `H` for the handler clauses in `asList`.

What type should `asList` have? Naively, we might simply expect to ignore the handler:

```
asList : [](() → 1) → List Int)
```

This would be unsound as it would allow us to write:

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

The function passed to `asList` yields a string. This is then accidentally handled by the handler in `asList`, which expects an integer.

A possible fix is to box the argument of `asList` with an absolute modality `[Gen Int]`:

```
asList : []([Gen Int](1 → 1) → List Int)
```

To see what happens here, consider the following typing judgement for the inlined function body of `asList` under some effect context `E`.

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

The effect handler extends the ambient effect context `E` with a `yield` operation to give an effect context of `Gen Int, E`. Meanwhile, the argument `f` has the effect context `Gen Int` specified by the absolute modality `[Gen Int]`. This is sound, because it is safe to invoke a function which can only use `Gen Int` under the effect context `Gen Int, E`.

However, the restriction that the argument can only use `Gen Int` severely hinders reusability. We would like to apply `asList` to arguments that may perform other operations in addition to `yield`.

To this end, we introduce *relative modalities* which enable us to describe the relative change that a handler makes to the effect context. For instance, consider:

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

The relative modality `<Gen Int>` is part of the argument type and extends the ambient effect context with `Gen Int` for the inner function `1 → 1`. The typing judgement becomes:

```
⊢ fun f → handle f () with H : <Gen Int>( 1 → 1 ) → List Int @ E
      f      @ Gen Int, E
      f ()   @ Gen Int, E
      1 → 1  @ Gen Int, E
```

Now, the effect context for the function of argument `f` is also `Gen Int, E`, matching the effect context at its invocation. This allows the argument `f` to perform other effects from the ambient effect context `E` (which will be forwarded to outer handlers).

In practice relative modalities often appear in an argument position and specify which effects of an argument will be handled in the function body. A higher-order function that handles effects `D` of its argument typically has a type of the form `<D>(1 → A) → B`.

In a traditional row-based effect system, in order to be able to use `asList` across different effect contexts, we would typically require effect polymorphism [22, 32].

```
asList : ∀ e . (1 --Gen Int, e--> 1) -e-> List Int
```

2.4 Coercions between Modalities

The implicit unboxing and boxing performed by Metl allows values to be coerced between different modalities. For instance, we can extend an absolute modality:

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

Not all modalities can be coerced to one another. For example, we cannot extend a relative modality

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

as doing so would insert a fresh `yield : Int ↠ 1` operation which may shadow other `yield` operations in `E`, consequently permitting bad programs like `crash` in Section 2.3.

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

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

However, the converse is not permitted

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

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

Similarly, the following typing judgement is invalid

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

because the argument `f` may use `Gen Int` in addition to the ambient effect context `E`.

2.5 Composing Handlers

We can compose handlers modularly. For example, consider state operations `get` and `put`.

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

Specialising the state for integers, we can give the standard state-passing interpretation of `State Int` as follows [43].

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

Using integer state we can write a generator which yields the prefix sum of a list.

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

The absolute modality `[Gen Int, State Int]` aggregates all effects performed in `prefixSum`.

We can now handle `prefixSum` by composing two handlers in sequence.

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

The type signature of `state` mentions only `State Int` even though it is applied to a computation which invokes `prefixSum`, which also uses `Gen Int`. In contrast, to achieve the same modularity, conventional row-based effect systems would ascribe the following type to `state`:

```
state : ∀ e . (1 -{State Int, e}-> 1) -e-> Int -e-> 1
```

2.6 Storing Effectful Functions in Data Types

We show how modal effect types allow us to smoothly store effectful functions into data types. We consider a richer effect handler example that implements cooperative concurrency using a UNIX-style fork operation [25, 46]. A `Coop` effect context includes two operations.

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

The `ufork` operation returns a boolean. As we shall see, concurrency can be implemented by a handler that invokes the continuation twice. The idea is that passing true to the continuation defines the behaviour of the parent, whereas passing false defines the behaviour of the child. The `suspend` operation suspends the current process allowing another process to run.

We model a process as a data type that embeds a continuation function which takes a list of suspended processes and returns unit. In addition, we define auxiliary functions `push` to append a process onto the end of the list and `next` to remove and then run the process at the head of the list.

```
data Proc = proc (List Proc → 1)

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

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

The following handler implements a scheduler parameterised by a list of suspended processes.

```
schedule : [](<Coop>(1 → 1) → List Proc → 1)
schedule m = handle m () with
  return ()             ↦ fun ps → next ps
  (suspend : 1↠1)  () r ↦ fun ps → next (push (proc (fun ps' → r () ps')) ps)
  (ufork : 1↠Bool) () r ↦ fun ps → r true (push (proc (fun ps' → r false ps')) ps)
```

The **`return`**-case is triggered when a process finishes, and runs the next available process. The `suspend`-case pushes the continuation onto the end of the list, before running the next available process. The `ufork`-case implements the process duplication behaviour of UNIX fork by first pushing

one application of the continuation onto the end of the list, and then immediately applying the other. Observe that the above code seamlessly stores continuation functions in `Proc` and then puts `Proc` in `List` without even mentioning any effects. These functions are not restricted to be pure; they may use any effects from the ambient effect context.

The `schedule` function allows processes to use any other effects. To achieve this flexibility, a traditional row-based effect system requires effect polymorphism and a parameterised data type.

```
data Proc e = proc (List Proc -e-> 1)
schedule : ∀ e . (1 -Coop, e-> 1) -e-> List (Proc e) -e-> 1
```

2.7 Masking

Whereas handlers extend the effect context, masking restricts the effect context [5]. Masking is a useful device to conceal private implementation details [35]. We illustrate masking by using a generator to implement a function to find an integer satisfying a predicate.

```
findWrong : [](Int → Bool) → List Int → Maybe Int)  # ill-typed
findWrong p xs = handle (iter (fun x → if p x then do yield x) xs) with
  return _              ↦ nothing
  (yield : Int ↠ 1) x _ ↦ just x
```

The `findWrong` program is ill-typed because it is unsound to invoke predicate `p` inside the handler, as this would accidentally handle any `yield` operations performed by `p`.

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

Changing the type of `p` from `Int → Bool` to `<Gen Int>(Int → Bool)` would fix the type error but leak the implementation detail that `findWrong` uses `yield`. A better solution is to mask `yield` for the argument `p` and rewrite the handled expression as follows.

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

The term **`mask`**`<yield>(M)` masks the operation `yield` from the ambient context of the subterm `M`. In doing so, it conceals any `yield` invocations of `M` from its immediate enclosing handler, thus deferring handling of those operations to the second-nearest dynamically enclosing handler. Now, the effect context for `p` is equivalent to the ambient context `E`, since the transformations of extending with `yield` (performed by the immediate enclosing handler) followed by masking with `yield` (performed by the mask) cancel each other out. Like a handler, a mask wraps its return value in a relative modality. The term **`mask`**`<yield>(p x)` initially returns a value of type `<yield|>Bool` instead of `Bool`, where `<yield|>` is a relative modality masking `yield` from the ambient context. Metl automatically unboxes the relative modality, because pure values (e.g. booleans) are oblivious of the effect context.

In general, relative modalities have the form `<L|D>` which specifies a local transformation on the effect context. Here, `L` represents the row of effect labels that are being removed from the context, whilst `D` represents the row of effects being added to it. We write `<D>` as a shorthand for `<|D>`.

2.8 Kinds

A handler extends the effect context with those effects it handles. When a value leaves the scope of a handler, its effect context changes, and we must keep track of this change. This requires the introduction of a simple kinding system.

For example, let us consider `state'`, a variation of the `state` function defined in Section 2.5 in which the return type of the handled computation is changed from `1` to `1 → 1`. The body of `state'` is exactly the same as that of `state`. We might naively expect its type signature to be the following.

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

However, this typing is not sound. Suppose we apply `state'` as follows.

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

The function `fun () → do put (do get () + 42)` is returned by the return clause of `state'`, escaping the scope of their handler. To guarantee effect safety, we must capture the fact that the returned function is a thunk which might perform `get` and `put` when invoked. The following typing is sound.

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

Let us contrast the types of `state : [](<State Int>(1 → 1) → Int → 1` and `state'`. The crucial difference is that the former cannot leak the state effect as the handled computation has unit type, whereas the latter can as the handled computation is a function.

In practice, it is useful to allow a value of base type or an algebraic data type that contains only base types or types boxed with absolute modalities to appear anywhere, including escaping the scope of a handler. Such values can never depend on the effect context in which they are used. We introduce a kind system in which the `Abs` kind classifies such *absolute types*, whereas the `Any` kind classifies unrestricted types. Subkinding allows absolute types to be treated as unrestricted.

2.9 Polymorphism for Value Types

Now that we have explored the simply-typed fragment of modal effect types, we briefly outline its extension with polymorphism for value types. For simplicity, Metl requires explicit type abstraction and type application. We write explicit type abstractions and applications using braces. For instance, we can define the polymorphic iterate function as follows.

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

The extension is mostly routine, however, we must respect kinds. The `state` and `state'` examples in Section 2.8 illustrate a non-uniformity that we must account for. We may generalise them such that the former allows any absolute return type and the latter allows any return type at all.

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

The syntax `∀ [a]` ascribes kind `Abs` to `a`, allowing values of type `a` to escape the handler. The syntax `∀ a` ascribes kind `Any` to `a`, not allowing values of type `a` to escape the handler. Though in practice it is usually desirable for return types of computations inside handler scopes to be absolute. The latter type signature is the more general in that simply by η-expanding we can coerce it to the former.

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

2.10 Effect Polymorphsim

Though modal effect types alone suffice for writing a remarkably rich class of modular effectful programs, occasionally effect variables are still useful. In particular, they are required for the implementation of higher-order effects [4, 53, 54, 57], which take closures as arguments.

Modal effect types restrict the parameter type and result type of an operation to be absolute. This is because effect handlers provide non-trivial manipulation of control-flow, which allows the

parameter and result of an operation to jump between different effect contexts. For example, if we were to allow an operation `leak : (1→1) ↠ 1`, then we could write the following unsafe program.

```
handle asList (fun () → do leak (fun () → do yield 42)) with
  return                _   ↦ fun () → 37
  (leak : (1 → 1) ↠ 1) p _ ↦ p
```

The `asList` handler extends the ambient effect context with `yield`. However, the `leak` handler binds the closure `(fun () → do yield 42)` to `p` and returns this closure, leaking the `yield` operation.

Consider then a higher-order fork operation whose parameter is a thunk (this operation is realisable with one-shot continuations; whereas the fork operation of Section 2.6 requires multi-shot continuations). We may define a recursive effect context for cooperative processes as follows.

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

This is sound because the parameter of `fork` is under an absolute modality. However, this stops the forked process from using any effect other than the two concurrency primitives `fork` and `suspend`. To allow it to use other effects, we re-introduce effect variables and polymorphism. With an effect variable `e`, we can define the following higher-order `Coop` parameterised over effect context `e`.

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

In Section 6.2 we show that the extension of effect variables is sound and backward compatible. Nonetheless, effect variables are only necessary for use-cases such as higher-order effects in which a computation needs to be stored for use in an effect context different from the ambient one.

3 A Tale of Locks and Keys: Elaborating MetL into Met

So far, we have presented a series of examples in MetL illustrating the core ideas of modal effect types. While MetL is an easy-to-use surface language, it contains too many implicit coercions to be a suitable basis for a core calculus. Instead, our core calculus Met is a more explicit language based on (simply-typed) multimodal type theory (MTT) [18, 19]. In this section, we motivate the design of Met and introduce core concepts of MTT. For a more detailed account of the simply-typed fragment of MTT, we refer the reader to the work of Kavvos and Gratzer [30].

MTT extends type theory with the notions of *modes* and *modalities* and is parametric in them. A typing judgement has the form $\Gamma \vdash M : A \mathbin{@} E$, which means that term M has type A under context Γ at mode E. In this work we consider *effect contexts* as modes, and use *absolute* modalities `[E]` and *relative* modalities `<L|D>` to move between them. In MTT, most type constructors are mode-local: the components have the same mode as the whole type. For example, if a function type $A \longrightarrow B$ is at mode E, then both A and B are at mode E. This is exactly the behaviour needed for effect contexts in Met (Section 2.1), and is a primary motivation for treating effect contexts as modes in Met.

In MTT, modes can be transformed by modalities. A modality $\mu : E \longrightarrow F$ transforms types and terms from mode E to mode F. While this is implicit in MetL, Met requires explicit terms for modality introduction and elimination. We write $\mathbf{mod}_{\mu}$ for introducing the modality μ. For example, the function `fun x → do yield x` of type `[yield : Int ↠ 1](Int → 1)` from Section 2.2 is elaborated to the following term in Met which explicitly introduces the absolute modality.

$$\mathbf{mod}_{[\texttt{yield:Int} \twoheadrightarrow 1]}\ (\lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x)$$

In order to invoke this function, we need to eliminate its modality first. In MetL, we need not manually unbox variables; elaboration does so for us. Met adopts let-style unboxing, which requires binding a term in order to unbox it. For example, consider that we bind the above function to the variable `gen` and then apply it to 42. The binding and application are elaborated to Met as follows.

$$\mathbf{let}\ \mathbf{mod}_{[\texttt{yield:Int} \twoheadrightarrow 1]}\ \texttt{gen} = \mathbf{mod}_{[\texttt{yield:Int} \twoheadrightarrow 1]}\ (\lambda x^{\texttt{Int}}.\mathbf{do}\ \texttt{yield}\ x)\ \mathbf{in}\ \texttt{gen}\ 42$$

The let-binding eliminates the absolute modality. Whenever `gen` is used, the type system ensures that the effect context contains at least the operation $\texttt{yield} : \texttt{Int} \twoheadrightarrow 1$. Both boxing and unboxing interact with the typing context non-trivially. Their typing rules in MTT are as follows[1].

$$\frac{\mu : E \to F \qquad \Gamma, \text{🔒}_\mu \vdash M : A \;@\; E}{\Gamma \vdash \textbf{mod}_\mu\ M : \mu A \;@\; F} \qquad \frac{\Gamma \vdash M : \mu A \;@\; E \qquad \Gamma, x :_\mu A \vdash N : B \;@\; E}{\Gamma \vdash \textbf{let mod}_\mu\ x = M \textbf{ in } N : B \;@\; E}$$

Contexts Γ are ordered. Boxing is mediated by a locked context. Unboxing binds a variable annotated with the corresponding modality. These locks and annotations are crucial for controlling variable access. For instance, consider the following typing judgement

$$x : \mu A \vdash \textbf{let mod}_\mu\ x' = x \textbf{ in } \textbf{mod}_\nu\ x' : \nu A \;@\; F$$

which unboxes a variable of type μA and re-boxes it with modality ν. If this term is typable, then its typing derivation would contain the following judgement for variable x':

$$x : \mu A, x' :_\mu A, \text{🔒}_\nu \vdash x' : A \;@\; E \quad \text{where } \nu : E \to F$$

Whether this usage of x' is valid or not depends on the mode theory. Beyond modes and modalities, a mode theory also specifies a set of modality transformations $\alpha : \mu \Rightarrow \nu$. (For those readers familiar with category theory, the structure of modes, modalities, and transformations forms a 2-category.) We can use a variable $x' :_\mu A$ across a lock 🔒_ν only if there exists a transformation $\alpha : \mu \Rightarrow \nu$.

In fact, the Met kind system relaxes this constraint slightly by allowing variables of absolute kind to cross locks even when such a transformation does not exist. Nonetheless, constraints on the form of modality transformations in Met (Section 4.3) are crucial for disallowing unsound coercions, between certain relative modalities, for instance. A case in point is the second example of Section 2.4, where we saw that Metl disallows coercing a function of type `<>(1 -> 1)` into a function of type `<Gen Int>(1 -> 1)`. This example is elaborated into Met as follows:

$$\lambda f^{\langle\rangle(1\to 1)}.\textbf{let mod}_{\langle\rangle}\ \hat{f} = f \textbf{ in } \textbf{mod}_{\langle|\texttt{Gen Int}\rangle}\ \hat{f} : \langle|\texttt{Gen Int}\rangle(1 \to 1) \;@\; E$$

First, f is unboxed to obtain $\hat{f} :_{\langle\rangle} (1 \to 1)$ in the context. Then, it is boxed again with the $\langle|\texttt{Gen Int}\rangle$ modality. However, just as in Metl, this example is not well-typed in Met. The reason for this is that the term $\textbf{mod}_{\langle|\texttt{Gen Int}\rangle}$ introduces a lock $\text{🔒}_{\langle|\texttt{Gen Int}\rangle}$ in the context. The variable $\hat{f}$ can only be used under the lock if there is a modality transformation of form $\langle\rangle \Rightarrow \langle|\texttt{Gen Int}\rangle$. But as explained in Section 2.4, such a transformation would break type safety and is thus not permitted.

Locks are introduced in the context whenever a typing rule changes the effect context. This happens not only during boxing but also in the rules for handlers and masks. For example, the definition of `asList : [](<Gen Int>(1 -> 1) -> List Int)` from Section 2.3 is elaborated to

$$\textbf{mod}_{[]}\ (\lambda f^{\langle|\texttt{Gen Int}\rangle(1\to 1)}.\textbf{let mod}_{\langle|\texttt{Gen Int}\rangle}\ \hat{f} = f \textbf{ in handle } \hat{f}\ () \textbf{ with } \hat{H})$$

where $\hat{H}$ is the elaboration of handler clauses H. The handler for the `Gen` effect introduces a lock $\text{🔒}_{\langle|\texttt{Gen Int}\rangle}$ in the context. We can use the variable $\hat{f}$ under the lock if there is a modality transformation $\alpha : \langle|\texttt{Gen Int}\rangle \Rightarrow \langle|\texttt{Gen Int}\rangle$. This identity transformation always exists.

4 A Multimodal Core Calculus with Effect Handlers

In this section we introduce Met, a simply-typed call-by-value calculus with effect handlers and modal effect types. We present its static and dynamic semantics as well as its meta theory. We aim at a minimal core calculus here and defer extensions such as data types, alternative forms of handlers, and polymorphism (including both for values and effects) to Section 6.

[1]As we will see in Section 4, in Met modalities on bindings and locks have indexes and the $\textbf{let mod}_\mu$ syntax also has an additional annotation. We opt for a simplified version here to convey the core intuition.

4.1 Syntax

The syntax of Met is as follows.

Types	$A, B ::= 1 \mid A \to B \mid \mu A$	Contexts	$\Gamma ::= \cdot \mid \Gamma, x :_{\mu_F} A \mid \Gamma, \blacksquare_{\mu_F}$
Masks	$L ::= \cdot \mid \ell, L$	Terms	$M, N ::= () \mid x \mid \lambda x^A.M \mid M\,N \mid \mathbf{mod}_\mu\, V$
Extensions	$D ::= \cdot \mid \ell : P, D$		$\mid \mathbf{let}_\nu\, \mathbf{mod}_\mu\, x = V\, \mathbf{in}\, M$
Effect Contexts	$E, F ::= \cdot \mid \ell : P, E$		$\mid \mathbf{do}\ \ell\ M \mid \mathbf{mask}_L\, M$
Presence	$P ::= A \twoheadrightarrow B \mid -$		$\mid \mathbf{handle}\ M\ \mathbf{with}\ H$
Modalities	$\mu, \nu ::= [E] \mid \langle L \mid D \rangle$	Values	$V, W ::= () \mid x \mid \lambda x^A.M \mid \mathbf{mod}_\mu\, V$
Kinds	$K ::= \mathsf{Abs} \mid \mathsf{Any}$	Handlers	$H ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto M\} \uplus H$

Met extends a simply-typed λ-calculus with standard constructs for effects and handlers as well as the main novelty of this work: modal effect types. We highlight the novel parts in grey.

We have provided a brief introduction to MTT in Section 3. We present Met without assuming deep familiarity with MTT and discuss further in Section 8.4. MTT provides us with the flexibility to define our own mode theory. In the following, we first illustrate the structures of modes, modalities, and modality transformations for Met before presenting the typing rules.

4.2 Effect Contexts as Modes

The *modes* of Met are effect contexts E. Each type and term is at some effect context E, specifying the available effects from the context.

Effect contexts E are defined as scoped rows of effect labels [31]. Each label denotes an effectful operation. An effect context may contain the same label multiple times. Each label has a presence type P [45]. A presence type P can be an operation arrow of the form $A \twoheadrightarrow B$, which indicates that the operation takes an argument of type A and returns a value of type B, or absent $-$, which indicates that the operation of this label cannot be invoked.

Following Rémy [45] and Leijen [31], we identify effects up to reordering of distinct labels, and allow absent labels to be freely added to or removed from the right of effect contexts. For instance, $\ell : P, \ell' : -$ is equivalent to $\ell : P$. We can think of an effect context as denoting a map from labels to infinite sequences of presence types where a cofinite tail of each sequence contains only $-$.

Extensions D and masks L are used respectively to extend effect contexts with more labels or removes some labels from them. Extensions are like effect contexts except that we do not ignore labels with absent types in their equivalence relation, so $\ell : P, \ell' : -$ and $\ell : P$ are distinct.

We define a sub-effecting relation on effect contexts $E \leqslant E'$ if we can replace the absent types in E with proper operation arrows to obtain E'. We also have a subtyping relation on extensions $D \leqslant D'$. Like sub-effecting on effect contexts, it requires D and D' to contain the same row of labels, but it allows absent types in D to be replaced by concrete signatures in D'. We give the full rules for type equivalence and sub-effecting in Appendix A.3.

Masks L are simply multisets of labels without presence types; we only need labels when removing them from effect contexts. We define three operations $D + E$, $E - L$, and $L \bowtie D$ as follows.

$$D + E = D, E \qquad \cdot - L = \cdot \qquad (\ell : P, E) - L = \begin{cases} E - L' & \text{if } L \equiv \ell, L' \\ \ell : P, (E - L) & \text{otherwise} \end{cases}$$

$$L \bowtie \cdot = (L, \cdot) \qquad L \bowtie (\ell : P, D) = \begin{cases} L' \bowtie D & \text{if } L \equiv \ell, L' \\ (L', (\ell : P, D')) & \text{otherwise} \\ \quad \text{where } (L', D') = L \bowtie D \end{cases}$$

The operation $D + E$ extends E with D. The operation $E - L$ removes the labels in L from E. The operation $L \bowtie D = (L', D')$ gives the difference between L and D. The L' are those labels in L not appearing in the domain of D, and the D' are those entries in D with labels not in L.

4.3 Modalities Manipulating Effect Contexts

Components of types and terms may have different effect contexts from the ambient one. We use *modalities* to manipulate effect contexts. For the modal type μA, the effect context for A is derived from the ambient effect context manipulated by the modality μ as follows.

$$[E](F) = E \qquad\qquad \langle L|D\rangle(F) = D + (F - L)$$

The absolute modality $[E]$ completely replaces the effect context F with E, similar to effect annotations on function types in traditional effect systems. The relative modality $\langle L|D\rangle$ is the key novelty of Met. It specifies a transformation on the input effect context. It masks the labels L in F before extending the resulting context with D. We call $\langle|\rangle$ the identity modality and write $\langle\rangle$ for it. Modalities are monotone total functions on effect contexts. If $E \leqslant F$, we have $\mu(E) \leqslant \mu(F)$.

We write μ_F for the pair of μ and F where F is the effect context that μ acts on. We refer to such a pair as a concrete modality. We write $\mu_F : E \longrightarrow F$ if $\mu(F) = E$. The arrow goes from E to F instead of the other direction to be consistent with MTT. Note that our terminology here differs slightly from that of MTT introduced in Section 3. Concrete modalities μ_F correspond to the notion of modalities in MTT and our modalities μ are actually indexed families of modalities in MTT.

Modality Composition. We can compose the actions of modalities in the intuitive way.

$$\begin{array}{rcll} \mu & \circ & [E] & = & [E] \\ [E] & \circ & \langle L|D\rangle & = & [D + (E - L)] \\ \langle L_1|D_1\rangle & \circ & \langle L_2|D_2\rangle & = & \langle L_1 + L|D_2 + D\rangle \quad \text{where } (L, D) = L_2 \bowtie D_1 \end{array}$$

To keep close to MTT, our composition reads from left to right. First, an absolute modality completely specifies the new effect context, thus shadowing any other modality μ. Second, replacing the effect context with E and then masking L and extending with D is equivalent to just replacing with $D + (E - L)$. Third, sequential masking and extending can be combined into one by using $L_2 \bowtie D_1$ to cancel the overlapping part of L_2 and D_1. For instance, we have $\langle\!|\texttt{yield} : \texttt{Int} \twoheadrightarrow 1\rangle \circ \langle \texttt{yield} |\!\rangle = \langle\rangle$.

Composition is well-defined since composing followed by applying is equivalent to sequentially applying $(\mu \circ \nu)(E) = \nu(\mu(E))$. We also have associativity $(\mu \circ \nu) \circ \xi = \mu \circ (\nu \circ \xi)$ and identity $\langle\rangle$. The definition of composition naturally generalises to concrete modalities μ_F. We can compose $\mu_F : E \longrightarrow F$ and $\nu_E : E' \longrightarrow E$ to get $\mu_F \circ \nu_E : E' \longrightarrow F$ which is defined as $(\mu \circ \nu)_F$.

Modality Transformations. Just as modalities allow us to manipulate effect contexts, we need a transformation relation that tells us when we can change modalities.

In Met, there could only be at most one transformation between any two modalities. As a result, we do not need to give names to transformations. We write $\mu_F \Rightarrow \nu_F$ for a transformation between concrete modalities $\mu_F : E \longrightarrow F$ and $\nu_F : E' \longrightarrow F$. Intuitively, such a transformation indicates that under ambient effect context F, the action of μ can be replaced by the action of ν. This relation is used to control variable access as we have demonstrated in Section 3. For instance, supposing we have a variable of type $\mu(1 \longrightarrow 1)$ under ambient effect context F, we can rewrap it to a function of type $\nu(1 \longrightarrow 1)$ if $\mu_F \Rightarrow \nu_F$.

Intuitively, $\mu_F \Rightarrow \nu_F$ is safe when $\nu(F)$ is larger than $\mu(F)$ so that we have not lost any operations. Moreover, subeffecting should not break the safety guarantee of transformations. That is, $\mu(F') \leqslant \nu(F')$ should hold for any effect context F' with $F \leqslant F'$. We formally define $\mu_F \Rightarrow \nu_F$ by the transitive closure of the following four rules.

MT-Abs
$$\frac{\mu_F : E' \longrightarrow F \quad E \leqslant E'}{[E]_F \Rightarrow \mu_F}$$

MT-Upcast
$$\frac{D \leqslant D'}{\langle L|D\rangle_F \Rightarrow \langle L|D'\rangle_F}$$

MT-Expand
$$\frac{(F - L) \equiv \ell : A \twoheadrightarrow B, E}{\langle L|D\rangle_F \Rightarrow \langle \ell, L|D, \ell : A \twoheadrightarrow B\rangle_F}$$

MT-Shrink
$$\frac{(F - L) \equiv \ell : P, E}{\langle \ell, L|D, \ell : P\rangle_F \Rightarrow \langle L|D\rangle_F}$$

MT-Abs allows us to transform an absolute modality to any other modality as long as no effect leaks. MT-Upcast allow us to upcast a label with an absent type in D to an arbitrary presence type, since the corresponding operation is unused. Recall that the subtyping relation between extensions only upcasts presence types. MT-Expand allows us to simultaneously mask and extend some present operations given that these operations exist in the ambient effect context F. MT-Shrink allows us to do the reverse for any operations regardless of their presence.

The following lemma shows that the syntactic definition of transformation matches our intuition. The proof is in Appendix B.2.

Lemma 4.1 (Semantics of modality transformation). *We have $\mu_F \Rightarrow \nu_F$ if and only if $\mu(F') \leqslant \nu(F')$ for all F' with $F \leqslant F'$.*

Let us give some examples. First, $[]_E \Rightarrow \mu_E$ always holds, consistent with the intuition that pure values can be used anywhere safely. Second, $\langle\!|\ell : -\rangle_E \Rightarrow \langle\!|\ell : P\rangle_E$ always holds. Third, we have $\langle \ell | \ell : A \twoheadrightarrow B\rangle_{\ell:A\twoheadrightarrow B,E} \Leftrightarrow \langle\rangle_{\ell:A\twoheadrightarrow B,E}$ in both directions. Last, $\langle\rangle_E \Rightarrow \langle\!|\ell : P\rangle_E$ does not hold for any E.

4.4 Kinds and Contexts

$\boxed{\Gamma \vdash A : K}$ $\boxed{\Gamma \vdash P}$ $\boxed{\Gamma \vdash (\mu, A) \Rightarrow \nu \mathbin{@} F}$

$$\frac{}{\Gamma \vdash 1 : \mathsf{Abs}} \qquad \frac{\Gamma \vdash A : \mathsf{Abs}}{\Gamma \vdash A : \mathsf{Any}} \qquad \frac{\Gamma \vdash [E] \qquad \Gamma \vdash A : \mathsf{Any}}{\Gamma \vdash [E]A : \mathsf{Abs}} \qquad \frac{\Gamma \vdash \langle L | D\rangle \qquad \Gamma \vdash A : K}{\Gamma \vdash \langle L | D\rangle A : K}$$

$$\frac{\Gamma \vdash A : \mathsf{Any} \quad \Gamma \vdash B : \mathsf{Any}}{\Gamma \vdash A \to B : \mathsf{Any}} \qquad \frac{\Gamma \vdash A : \mathsf{Abs} \quad \Gamma \vdash B : \mathsf{Abs}}{\Gamma \vdash A \twoheadrightarrow B} \qquad \frac{\Gamma \vdash A : \mathsf{Abs}}{\Gamma \vdash (\mu, A) \Rightarrow \nu \mathbin{@} F} \qquad \frac{\mu_F \Rightarrow \nu_F}{\Gamma \vdash (\mu, A) \Rightarrow \nu \mathbin{@} F}$$

$\boxed{\Gamma \mathbin{@} E}$

$$\frac{}{\cdot \mathbin{@} E} \qquad \frac{\Gamma \mathbin{@} F \qquad \mu_F : E \to F \qquad \Gamma \vdash A : K}{\Gamma, x :_{\mu_F} A \mathbin{@} F} \qquad \frac{\Gamma \mathbin{@} F \qquad \mu_F : E \to F}{\Gamma, \text{🔒}_{\mu_F} \mathbin{@} E}$$

Fig. 1. Representative kinding, well-formedness, and auxiliary rules for Met.

As illustrated in Section 2.8, we have two kinds Abs and Any. The Abs kind is a sub-kind of the kind of all types Any, and denotes types of values that are guaranteed not to use operations from the ambient effect context. We show the kinding and well-formedness rules for types and presence types in Figure 1, relying on the well-formedness of modalities $\Gamma \vdash \mu$ and effect contexts $\Gamma \vdash E$, which is standard and defined in Appendix A.3. Function arrows have kind Any due to the possibility of using operations from the ambient effect context. A modal type $[E]A$ is absolute as it cannot depend on the ambient effect context. We restrict the kind of the argument and return value of effects to be Abs in order to prevent effect leakage as discussed in Section 2.10.

Contexts are ordered. Each term variable binding $x :_{\mu_F} A$ in contexts is tagged with an concrete modality μ_F. We omit this annotation when μ is identity. Contexts contain locks 🔒_{μ_F} carrying concrete modalities μ_F. As shown in Section 3, they track the introduction and elimination of modalities and play an important role in controlling variable access. We omitted indexes of modalities on bindings and locks in Section 3 for brevity; they are obvious from the context.

We define the relation $\Gamma \mathbin{@} E$ that context Γ is well-formed at effect context E in Figure 1. For instance, given some modalities $\mu_F : E_1 \to F$, $\nu_F : E_2 \to F$, and $\xi_E : E_3 \to E$, the following context

is well-formed at effect context E. Reading from left to right, the lock $🔒_{[E]_F}$ switches the effect context from F to E as $[E](F) = E$.

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

Following MTT, we define $\mathsf{locks}(-)$ to compose all the modalities on the locks in a context.

$$\mathsf{locks}(\cdot) = \langle\rangle \qquad \mathsf{locks}(\Gamma, 🔒_{\mu_F}) = \mathsf{locks}(\Gamma) \circ \mu_F \qquad \mathsf{locks}(\Gamma, x :_{\mu_F} A) = \mathsf{locks}(\Gamma)$$

Following MTT, we identify contexts up to the following two equations.

$$\Gamma, 🔒_{\langle\rangle_E} \ @\ E = \Gamma \ @\ E \qquad\qquad \Gamma, 🔒_{\mu_F}, 🔒_{\nu_{F'}} \ @\ E = \Gamma, 🔒_{\mu_F \circ \nu_{F'}} \ @\ E$$

4.5 Typing

The typing rules of Met are shown in Figure 2. The typing judgement $\Gamma \vdash M : A \ @\ E$ means that the term M has type A under context Γ and effect context E. As usual, we require $\Gamma \ @\ E$, $\Gamma \vdash E$, $\Gamma \vdash A : K$ for some K, and well-formedness for type annotations as well-formedness conditions. We explain the interesting rules, which are highlighted in grey; the other rules are standard.

$\boxed{\Gamma \vdash M : A \ @\ E}$

T-Var
$$\frac{\nu_F = \mathsf{locks}(\Gamma') : E \to F \qquad \Gamma \vdash (\mu, A) \Rightarrow \nu \ @\ F}{\Gamma, x :_{\mu_F} A, \Gamma' \vdash x : A \ @\ E}$$

T-Mod
$$\frac{\mu_F : E \to F \qquad \Gamma, 🔒_{\mu_F} \vdash V : A \ @\ E}{\Gamma \vdash \mathbf{mod}_\mu\, V : \mu A \ @\ F}$$

T-Letmod
$$\frac{\nu_F : E \to F \qquad \Gamma, 🔒_{\nu_F} \vdash V : \mu A \ @\ E \qquad \Gamma, x :_{\nu_F \circ \mu_E} A \vdash M : B \ @\ F}{\Gamma \vdash \mathbf{let}_\nu\ \mathbf{mod}_\mu\ x = V\ \mathbf{in}\ M : B \ @\ F}$$

T-Abs
$$\frac{\Gamma, x : A \vdash M : B \ @\ E}{\Gamma \vdash \lambda x^A.M : A \to B \ @\ E}$$

T-App
$$\frac{\Gamma \vdash M : A \to B \ @\ E \qquad \Gamma \vdash N : A \ @\ E}{\Gamma \vdash M\ N : B \ @\ E}$$

T-Do
$$\frac{E = \ell : A \twoheadrightarrow B, F \qquad \Gamma \vdash N : A \ @\ E}{\Gamma \vdash \mathbf{do}\ \ell\ N : B \ @\ E}$$

T-Mask
$$\frac{\Gamma, 🔒_{\langle L \rangle_F} \vdash M : A \ @\ F - L}{\Gamma \vdash \mathbf{mask}_L\ M : \langle L \rangle A \ @\ F}$$

T-Handler
$$\frac{\begin{array}{c} H = \{\mathbf{return}\ x \mapsto N\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \\ \Gamma, 🔒_{\langle\!| D \rangle\!\rangle_F} \vdash M : A \ @\ D + F \qquad \Gamma, x : \langle\!| D \rangle\!\rangle A \vdash N : B \ @\ F \\ D = \{\ell_i : A_i \twoheadrightarrow B_i\}_i \qquad [\Gamma, p_i : A_i, r_i : B_i \to B \vdash N_i : B \ @\ F]_i \end{array}}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : B \ @\ F}$$

Fig. 2. Typing rules for Met.

Modality Introduction and Elimination. Modalities are introduced by T-Mod and eliminated by T-Letmod. The term $\mathbf{mod}_\mu\, V$ introduces modality μ to the type of the conclusion and lock $🔒_{\mu_F}$ into the context of the premise, and requires the value V to be well-typed under the new effect context E manipulated by μ. The lock $🔒_{\mu_F}$ tracks the change to the effect context. Specialising the modality μ to either absolute or relative modalities, we get the following two rules.

$$\frac{\Gamma, 🔒_{[E]_F} \vdash V : A \ @\ E}{\Gamma \vdash \mathbf{mod}_{[E]}\, V : [E]A \ @\ F} \qquad\qquad \frac{\Gamma, 🔒_{\langle L | D \rangle_F} \vdash V : A \ @\ D + (F - L)}{\Gamma \vdash \mathbf{mod}_{\langle L | D \rangle}\, V : \langle L | D \rangle A \ @\ F}$$

Note that we use modality μ instead of concrete modality μ_F in types and terms, because the index can always be inferred from the effect context. We restrict **mod** to values to avoid effect

leakage [2, 34]. Otherwise, a term such as $\mathbf{mod}_{\langle\!|\,\ell:P\,\rangle}\ (\mathbf{do}\ \ell\ V)$ would type check under the empty effect context but get stuck due to the unhandled operation ℓ.

The term $\mathbf{let}_\nu\ \mathbf{mod}_\mu\ x = V\ \mathbf{in}\ M$ moves the modality μ from the type of V to the binding of x. As with boxing, unboxing is restricted to values. Following MTT, we use let-style modality elimination which takes another modality ν in addition to the modality μ that is eliminated from V. This is crucial for sequential unboxing. For instance, the following term sequentially unboxes $x : \nu\mu A$. The variables y and z are bound as $y :_\nu \mu A$ and $z :_{\nu\circ\mu} A$, respectively.

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

Masking and Handling. Masking and handling also introduce relative modalities. Unlike **mod**, these constructs can apply to computations as they perform masking and handling semantically. In T-Mask, the mask $\mathbf{mask}_L\ M$ removes effects L from the ambient effect context for M. For the return value of M, we need to box it with $\langle L \rangle$ to reconcile the mismatch between $F - L$ and F. In T-Handler, the handler $\mathbf{handle}\ M\ \mathbf{with}\ H$ extends the ambient effect context with effects D for M. For the return value of M which is bound as x in the return clause, we need to box it with $\langle\!|\, D \rangle$ to reconcile the mismatch between $D + F$ and F. The other parts of the handler rule are standard.

Accessing Variables. The T-Var rule uses the auxiliary judgement $\Gamma \vdash (\mu, A) \Rightarrow \nu\ @\ F$ defined in Figure 1. Variables of absolute types can always be used as they do not depend on the effect context. For a non-absolute term variable binding $x :_{\mu_F} A$ from context $\Gamma, x :_{\mu_F} A, \Gamma'$, we must guarantee that it is safe to use x in the current effect context. The term bound to x is defined inside μ under the effect context F. As we track all transformations on effect contexts up to the binding of x as locks in Γ', the current effect context E is obtained by applying $\mathsf{locks}(\Gamma')$ to F. Thus, we need the transformation $\mu_F \Rightarrow \mathsf{locks}(\Gamma')_F$ to hold for effect safety.

Subeffecting. Subeffecting is incorporated into the T-Var rule within the transformation relation $\mu_F \Rightarrow \nu_F$. We have seen how subeffecting works in Section 2.4. We give another example here which upcasts the empty effect context to E. It is well-typed because $[].\ \Rightarrow [E].$ holds.

$$\lambda x^{[]\,(\mathtt{Int}\to\mathtt{Int})}.\mathbf{let}\ \mathbf{mod}_{[]}\ y = x\ \mathbf{in}\ \mathbf{mod}_{[E]}\ y : []\,(\mathtt{Int} \to \mathtt{Int}) \to [E]\,(\mathtt{Int} \to \mathtt{Int})$$

4.6 Operational Semantics

The operational semantics for Met is quite standard [24]. We first define evaluation contexts $\mathcal{E}$:

$$\text{Evaluation contexts}\quad \mathcal{E} ::= [\,] \mid \mathcal{E}\ N \mid V\ \mathcal{E} \mid \mathbf{do}\ \ell\ \mathcal{E} \mid \mathbf{mask}_L\ \mathcal{E} \mid \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H$$

The reduction rules are as follows.

$$\begin{array}{lr}
\textsc{E-App} & (\lambda x^A.M)\ V \rightsquigarrow M[V/x] \\
\textsc{E-Letmod} & \mathbf{let}_\nu\ \mathbf{mod}_\mu\ x = \mathbf{mod}_\mu\ V\ \mathbf{in}\ M \rightsquigarrow M[V/x] \\
\textsc{E-Mask} & \mathbf{mask}_L\ V \rightsquigarrow \mathbf{mod}_{\langle L \rangle}\ V \\
\textsc{E-Ret} & \mathbf{handle}\ V\ \mathbf{with}\ H \rightsquigarrow N[(\mathbf{mod}_{\langle\!|\, D \rangle}\ V)/x], \text{where } (\mathbf{return}\ x \mapsto N) \in H \\
\textsc{E-Op} & \mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/p, (\lambda y.\mathbf{handle}\ \mathcal{E}[y]\ \mathbf{with}\ H)/r], \\
 & \qquad \text{where } 0\text{–free}(\ell, \mathcal{E}) \text{ and } (\ell\ p\ r \mapsto N) \in H \\
\textsc{E-Lift} & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \qquad \text{if } M \rightsquigarrow N
\end{array}$$

The only slightly non-standard aspect of the rules is the boxing of values escaping masks and handlers in E-Mask and E-Ret. They coincide with the typing rules for masks and handlers. In E-Ret, we assume handlers are decorated with the operations D that they handle as in Section 2.

Following Biernacki et al. [5], the predicate $n\text{–free}(\ell, \mathcal{E})$ is defined inductively on evaluation contexts as follows. We have $n\text{–free}(\ell, \mathcal{E})$ if there are n masks for ℓ in $\mathcal{E}$ without corresponding handlers. The E-Op rule requires $0\text{–free}(\ell, \mathcal{E})$ to guarantee that the current handler is not masked

by masks in $\mathcal{E}$. The meta function $\mathsf{count}(\ell; L)$ yields the number of ℓ labels in L. We omit the inductive cases that do not change n.

$$\frac{}{0\text{–}\mathsf{free}(\ell, [\,])} \qquad \frac{n\text{–}\mathsf{free}(\ell, \mathcal{E})}{n\text{–}\mathsf{free}(\ell, \mathbf{do}\ \ell'\ \mathcal{E})} \qquad \frac{n\text{–}\mathsf{free}(\ell, \mathcal{E}) \qquad \mathsf{count}(\ell; L) = m}{(n+m)\text{–}\mathsf{free}(\ell, \mathbf{mask}_L\ \mathcal{E})}$$

$$\frac{(n+1)\text{–}\mathsf{free}(\ell, \mathcal{E}) \qquad \ell \in \mathsf{dom}(H)}{n\text{–}\mathsf{free}(\ell, \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H)} \qquad \frac{n\text{–}\mathsf{free}(\ell, \mathcal{E}) \qquad \ell \notin \mathsf{dom}(H)}{n\text{–}\mathsf{free}(\ell, \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H)}$$

4.7 Type Soundness and Effect Safety

We prove type soundness and effect safety for Met. Our proofs cover the extensions in Section 6.

Met enjoys substitution properties along the lines of Kavvos and Gratzer [30]. For example, we have the following rule for substituting values with modalities into terms.

$$\frac{\Gamma, \blacksquare_{\mu_F} \vdash V : A \mathbin{@} F' \qquad \Gamma, x :_{\mu_F} A, \Gamma' \vdash M : B \mathbin{@} E}{\Gamma, \Gamma' \vdash M[V/x] : B \mathbin{@} E}$$

We state and prove the relevant properties in Appendix B.3.

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

Definition 4.2 (Normal Forms). We say a term M is in a normal form with respect to effect type E, if it is either a value V or of the form $M = \mathcal{E}[\mathbf{do}\ \ell\ V]$ for $\ell \in E$ and $n\text{–}\mathsf{free}(\ell, \mathcal{E})$.

The following together give type soundness and effect safety (proofs in Appendices B.4 and B.5).

Theorem 4.3 (Progress). *If $\vdash M : A \mathbin{@} E$, then either there exists N such that $M \rightsquigarrow N$ or M is in a normal form with respect to E.*

Theorem 4.4 (Subject Reduction). *If $\Gamma \vdash M : A \mathbin{@} E$ and $M \rightsquigarrow N$, then $\Gamma \vdash N : A \mathbin{@} E$.*

5 Encoding Effect Polymorphism in Met

Even without effect variables, Met is sufficiently expressive to encode programs from conventional row-based effect systems provided effect variables on function arrows always refer to the lexically closest one. This is an important special case, since most functions in practice use at most one effect variable. For example, as of July 2024, the Koka repository contains 520 effectful functions across 112 files but only 86 functions across 5 files use more than one effect variable, almost all of them internal primitives for handlers not exposed to programmers. Moreover, almost all programs in the Frank repository make no mention of effect variables, relying on syntactic sugar to hide the single effect variable. We formally characterise and prove this intuition on the expressiveness of Met.

5.1 Row Effect Types with a Single Effect Variable

We first define $\mathrm{F}^1_{\mathrm{eff}}$, a core calculus with row-based effect types in the style of Koka [32], but where each scope can only refer to the lexically closest effect variable.

$$\begin{array}{lll}
\text{Types} & A, B ::= 1 \mid A \to^{\{E \mid \varepsilon\}} B \mid \forall \varepsilon. A & \text{Terms} \quad M, N ::= () \mid x \mid \lambda^{\{E \mid \varepsilon\}} x^A.M \mid M\,N \\
\text{Effects} & L, D, E, F ::= \cdot \mid \ell, E & \qquad\qquad \mid \Lambda \varepsilon. V \mid M\,\#\{E \mid \varepsilon\} \mid \mathbf{do}\ \ell\ M \\
\text{Contexts} & \Gamma ::= \cdot \mid \Gamma, x :_\varepsilon A \mid \Gamma, \blacklozenge_E \mid \Gamma, \blacklozenge_E^\Lambda & \qquad\qquad \mid \mathbf{mask}_L\ M \mid \mathbf{handle}\ M\ \mathbf{with}\ H \\
\text{Values} & V, W ::= x \mid \lambda^{\{E \mid \varepsilon\}} x^A.M \mid \Lambda \varepsilon. V & \text{Handlers} \quad H ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto M\} \uplus H
\end{array}$$

In types we include units, effectful functions, and effect abstraction $\forall \varepsilon. A$. As we consider only one effect variable at a time, we need not track effect variables on function types and effect abstraction.

$\boxed{\Gamma \vdash M : A\,!\,\{E|\varepsilon\}}$

R-Var
$$\frac{\varepsilon = \varepsilon' \text{ or } A = \forall\varepsilon''.A' \text{ or } A = 1}{\Gamma_1, x :_{\varepsilon'} A, \Gamma_2 \vdash x : A\,!\,\{E|\varepsilon\}}$$

R-Abs
$$\frac{\Gamma, \blacklozenge_E, x :_{\varepsilon} A \vdash M : B\,!\,\{F|\varepsilon\}}{\Gamma \vdash \lambda^{\{F|\varepsilon\}} x^A.M : A \to^{\{F|\varepsilon\}} B\,!\,\{E|\varepsilon\}}$$

R-App
$$\frac{\Gamma \vdash M : A \to^{\{E|\varepsilon\}} B\,!\,\{E|\varepsilon\} \quad \Gamma \vdash N : A\,!\,\{E|\varepsilon\}}{\Gamma \vdash M\,N : B\,!\,\{E|\varepsilon\}}$$

R-EAbs
$$\frac{\varepsilon' \notin \mathrm{ftv}(\Gamma) \quad \Gamma, \blacklozenge_E^{\Lambda} \vdash V : A\,!\,\{\cdot|\varepsilon'\}}{\Gamma \vdash \Lambda\varepsilon'.V : \forall\varepsilon'.A\,!\,\{E|\varepsilon\}}$$

R-EApp
$$\frac{\Gamma \vdash M : \forall\varepsilon'.A\,!\,\{E|\varepsilon\}}{\Gamma \vdash M\,\#\{E|\varepsilon\} : A[\{E|\varepsilon\}/]\,!\,\{E|\varepsilon\}}$$

R-Mask
$$\frac{\Gamma, \blacklozenge_{L+E} \vdash M : A\,!\,\{E|\varepsilon\}}{\Gamma \vdash \mathbf{mask}_L\ M : A\,!\,\{L + E|\varepsilon\}}$$

R-Do
$$\frac{(\ell : A \twoheadrightarrow B) \in \Sigma \quad \Gamma \vdash M : A\,!\,\{\ell, E|\varepsilon\}}{\Gamma \vdash \mathbf{do}\ \ell\ M : B\,!\,\{\ell, E|\varepsilon\}}$$

R-Handler
$$\frac{\Gamma, \blacklozenge_E \vdash M : A\,!\,\{\overline{\ell_i}, E|\varepsilon\} \qquad \Gamma, x :_{\varepsilon} A \vdash N : B\,!\,\{E|\varepsilon\} \qquad \{\ell_i : A_i \twoheadrightarrow B_i\} \subseteq \Sigma \qquad [\Gamma, p_i :_{\varepsilon} A_i, r_i :_{\varepsilon} B_i \to^{\{E|\varepsilon\}} B \vdash N_i : B\,!\,\{E|\varepsilon\}]_i}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i : B\,!\,\{E|\varepsilon\}}$$

Fig. 3. Typing rules of $\mathrm{F}^1_{\mathrm{eff}}$.

Nonetheless, we include them in grey font for easier comparison with existing calculi. In Γ, each term variable is annotated with the effect variable ε that was referred to at the time of its introduction. Further, we add markers $\blacklozenge_E$ and $\blacklozenge_E^{\Lambda}$ to the context, which track the change of effects due to functions, masks, handlers, and effect abstraction. These markers are not needed by the typing rules but help with the encoding. As with Met, we require contexts to be ordered. For simplicity we assume operations are always present and defined by a global context $\Sigma = \{\overline{\ell : A \twoheadrightarrow B}\}$, thus unifying extensions D, masks L, and effect contexts E of Met into one syntactic category. Mirroring our kind restriction for operation arrows in Met, we assume that these A and B are not function arrows, but they can be effect abstractions (which may themselves contain function arrows).

Figure 3 gives the typing rules of $\mathrm{F}^1_{\mathrm{eff}}$. The judgement $\Gamma \vdash M : A\,!\,\{E|\varepsilon\}$ states that in context Γ, the term M has type A and might use concrete effects E extended with effect variable ε. The typing rules are mostly standard for row-based effect systems. The R-Var rule ensures that either the current effect variable matches the effect variable at which the variable was introduced or that the value is an effect abstraction or unit. These constraints guarantee that programs can only refer to one effect variable in one scope. The R-App, R-Do, R-Mask, and R-Handler rules are standard. The R-Abs rule is standard except for requiring the effect variable to remain unchanged. The R-EAbs rule introduces a new effect variable ε' and the R-EApp rule instantiates an effect abstraction. While conventional systems allow instantiating with any effect row, R-EApp only allows instantiation with the ambient effects E and effect variable ε. The instantiation operator $[\{E|\varepsilon\}/]$ implements standard type substitution for the single effect variable as follows.

$$1[\{E|\varepsilon\}/] = 1 \qquad (\forall\varepsilon'.A)[\{E|\varepsilon\}/] = \forall\varepsilon'.A$$
$$(A \to^{\{F|\varepsilon'\}} B)[\{E|\varepsilon\}/] = A[\{E|\varepsilon\}/] \to^{\{F,E|\varepsilon\}} B[\{E|\varepsilon\}/]$$

For example, the following $\mathrm{F}^1_{\mathrm{eff}}$ function (grey parts omitted) sums up all yielded integers.

$$\texttt{asSum} : \forall.(1 \to^{\texttt{Gen Int}} 1) \to \texttt{Int}$$
$$\texttt{asSum} = \Lambda.\lambda f.\mathbf{handle}\ f\ ()\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto 0, \texttt{yield}\ x\ r \mapsto x + r\ ()\}$$

5.2 Encoding

We now give compositional translations for types and contexts of $\mathrm{F}^1_{\mathrm{eff}}$ into Met. We transform $\mathrm{F}^1_{\mathrm{eff}}$ types at effect context E to modal types in Met by the translation $[\![-]\!]_E$.

$$\begin{array}{rcl} [\![1]\!]_E & = & \langle\rangle 1 \\ [\![A \to^F B]\!]_E & = & \langle E - F | F - E\rangle([\![A]\!]_F \to [\![B]\!]_F) \\ [\![\forall.A]\!]_E & = & [][\![A]\!]. \end{array} \qquad \begin{array}{rcl} [\![\cdot]\!]_E & = & \cdot \\ [\![\Gamma, x : A]\!]_E & = & [\![\Gamma]\!]_E, x :_{\mu_E} A' \text{ for } \mu A' = [\![A]\!]_E \\ [\![\Gamma, \blacklozenge_F]\!]_E & = & [\![\Gamma]\!]_F, \blacklozenge_{\langle F-E|E-F\rangle} \\ [\![\Gamma, \blacklozenge^{\Delta}_F]\!]. & = & [\![\Gamma]\!]_F, \blacklozenge_{[]} \end{array}$$

For the unit type, we insert the identity modality for the uniformity of the translation. For a function arrow $A \to^F B$, we use a relative modality $\langle E - F | F - E\rangle$ to wrap the function arrow. This modality transforms the outside effect context E to F which is specified by the original function arrow $A \to^F B$. For effect abstraction, we use an empty absolute modality to wrap the type, simulating entering a new scope with different effect variables. We translate contexts by translating each type and moving top-level modalities to their bindings. For each marker, we insert a corresponding lock to reflect the changes of effect context.

As an example of type translation, the type of `asSum` in $\mathrm{F}^1_{\mathrm{eff}}$ from Section 5.1 is translated to

$$\texttt{asSum} \quad : \quad [](\langle\!|\texttt{Gen Int}\rangle(\langle\rangle 1 \to \langle\rangle 1) \to \langle\rangle \texttt{Int}).$$

Observe that not every valid typing judgement in $\mathrm{F}^1_{\mathrm{eff}}$ can be transformed to a valid typing judgement in Met, because the translation depends on markers in contexts, while the typing of $\mathrm{F}^1_{\mathrm{eff}}$ does not. We define well-scoped typing judgements, which characterise the typing judgements for which our encoding is well-defined, as follows.

Definition 5.1 (Well-scoped). A typing judgement $\Gamma_1, x :_\varepsilon A, \Gamma_2 \vdash M : B \,!\, \{E|\varepsilon\}$ is *well-scoped for* x if either $x \notin \mathrm{fv}(M)$ or $\blacklozenge^{\Delta}_F \notin \Gamma_2$ or $A = \forall.A'$ or $A = 1$. A typing judgement $\Gamma \vdash M : A \,!\, \{E|\varepsilon\}$ is *well-scoped* if it is well-scoped for all $x \in \Gamma$.

In particular, if the judgement at the leaf of a derivation tree is well-scoped, then every judgement in the derivation tree is well-scoped. Also note that contexts with no markers are always well-scoped. Consequently, restricting judgements to well-scoped ones in $\mathrm{F}^1_{\mathrm{eff}}$ does not reduce expressive power, as we can always ensure that contexts at the leaves have no markers.

We write $M : A \,!\, E \dashrightarrow M'$ for the translation of a $\mathrm{F}^1_{\mathrm{eff}}$ term M of type A with effect context E to a Met term M'. We have the following type preservation theorem. The translation of terms and proof of type preservation can be found in Appendix C.

Lemma 5.2 (Type preservation of encoding). *If* $\Gamma \vdash M : A \,!\, \{E|\varepsilon\}$ *is well-scoped, then* $M : A \,!\, E \dashrightarrow M'$ *and* $[\![\Gamma]\!]_E \vdash M' : [\![A]\!]_E \mathbin{@} E$.

Intuitively, the term translation inserts boxing and unboxing constructs in appropriate places in order to realise transition between effect contexts as embodied by the type translation. Specifically, for variables we unbox them immediately after they are bound, and re-box them when they are used. As an example of term translation, we can translate the `asSum` handler in $\mathrm{F}^1_{\mathrm{eff}}$ defined in Section 5.1 as follows into Met, omitting boxing and unboxing of the identity modality $\langle\rangle$.

$$\begin{array}{l} \texttt{asSum} = \mathbf{mod}_{[]}(\lambda f.\mathbf{let\ mod}_{\langle\!|\texttt{Gen Int}\rangle}\ f = f\ \mathbf{in\ handle}\ f\ ()\ \mathbf{with} \\ \qquad \{\mathbf{return}\ x \mapsto \mathbf{let\ mod}_{\langle\!|\texttt{Gen Int}\rangle}\ x = x\ \mathbf{in}\ 0, \mathbf{yield}\ x\ r \mapsto x + r\ ()\}) \end{array}$$

The explicit boxing and unboxing, as well as the identity modalities, here are only generated to keep the encoding systematic. We need not write them in practice as illustrated in Section 2.

Our encoding focuses on presenting the core idea and does not consider advanced features including data types and polymorphism. In Section 6 we show how to extend Met with these features and in Appendix C.3 we briefly outline how to extend the encoding to cover them.

6 Extensions

In this section we demonstrate that Met scales to support data types and polymorphism including both value and effect polymorphism. Effect polymorphism helps deal with situations in which it is useful to refer to one or more effect contexts that differ from the ambient one (such as the higher-order fork operation in Section 2.10), recovering the full expressive power of row-based effect systems. We only discuss the key ideas of extensions here; their full specification as well as more extensions including shallow handlers [23, 27] are given in Appendix A. We prove type soundness and effect safety for all extensions in Appendix B.

6.1 Making Data Types Crisp

We demonstrate the extensibility of Met with data types by extending it with pair and sum types. We expect no extra challenge to extend Met with algebraic data types. The syntax and typing rules are shown as follows.

T-Pair
$$\frac{\Gamma \vdash M : A \mathbin{@} E \qquad \Gamma \vdash N : B \mathbin{@} E}{\Gamma \vdash (M, N) : A * B \mathbin{@} E}$$

T-Inl
$$\frac{\Gamma \vdash M : A \mathbin{@} E}{\Gamma \vdash \mathbf{inl}\ M : A + B \mathbin{@} E}$$

T-Inr
$$\frac{\Gamma \vdash M : B \mathbin{@} E}{\Gamma \vdash \mathbf{inr}\ M : A + B \mathbin{@} E}$$

T-CrispPair
$$\frac{\nu_F : E \to F \qquad \Gamma, \text{🔒}_{\nu_F} \vdash V : A * B \mathbin{@} E \qquad \Gamma, x :_{\nu_F} A, y :_{\nu_F} B \vdash M : A' \mathbin{@} F}{\Gamma \vdash \mathbf{case}_{\nu}\ V\ \mathbf{of}\ (x, y) \mapsto M : A' \mathbin{@} F}$$

T-CrispSum
$$\frac{\nu_F : E \to F \qquad \Gamma, \text{🔒}_{\nu_F} \vdash V : A + B \mathbin{@} E \qquad \Gamma, x :_{\nu_F} A \vdash M_1 : A' \mathbin{@} F \qquad \Gamma, y :_{\nu_F} B \vdash M_2 : A' \mathbin{@} F}{\Gamma \vdash \mathbf{case}_{\nu}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M_1, \mathbf{inr}\ y \mapsto M_2\} : A' \mathbin{@} F}$$

The T-Pair, T-Inl, and T-Inr are standard introduction rules. The elimination rules T-CrispPair and T-CrispSum are more interesting. In addition to normal pattern matching, they interpret the value V under the effect context transformed by certain modalities ν, which can then be tagged to the variable bindings in case clauses. They follow the crisp induction principles of multimodal type theory [19, 20, 47]. These crisp elimination rules provide extra expressiveness. For example, we can write the following function which transforms a sum of type $\mu(A + B)$ to another sum of type $(\mu A + \mu B)$. This function is not expressible without crisp elimination rules.

$$\lambda x^{\mu(A+B)}.\mathbf{let}\ \mathbf{mod}_{\mu}\ y = x\ \mathbf{in}\ \mathbf{case}_{\mu}\ y\ \mathbf{of}\ \{\mathbf{inl}\ x_1 \mapsto \mathbf{inl}\ (\mathbf{mod}_{\mu}\ x_1), \mathbf{inr}\ x_2 \mapsto \mathbf{inr}\ (\mathbf{mod}_{\mu}\ x_2)\}$$

6.2 Polymorphism for Values and Effects

The extensions to syntax and typing rules with polymorphism are as follows.

$$\begin{array}{lll} \text{Types} & A, B ::= \cdots \mid \forall \alpha^K.A & \qquad \text{Contexts} \quad \Gamma ::= \cdots \mid \Gamma, \alpha : K \\ \text{Effects} & E ::= \cdots \mid \varepsilon \mid E \backslash L & \qquad \text{Terms} \quad M, N ::= \cdots \mid \Lambda \alpha^K.V \mid M\,A \\ \text{Kinds} & K ::= \cdots \mid \text{Effect} & \qquad \text{Values} \quad V, W ::= \cdots \mid \Lambda \alpha^K.V \mid V\,A \end{array}$$

T-TAbs
$$\frac{\Gamma, \alpha : K \vdash V : A \mathbin{@} E}{\Gamma \vdash \Lambda \alpha^K.V : \forall \alpha^K.A \mathbin{@} E}$$

T-TApp
$$\frac{\Gamma \vdash M : \forall \alpha^K.B \mathbin{@} E \qquad \Gamma \vdash A : K}{\Gamma \vdash M\,A : B[A/\alpha] \mathbin{@} E}$$

It may appear surprising that we treat type application $V\,A$ as values. This is useful in practice to allow instantiation inside boxes. We also extend the semantics to allow reduction in values.

To support effect polymorphism, we extend the syntax of effect contexts E with effect variables ε and introduce a new kind Effect for them. As is typical for row polymorphism, we restrict each effect type to contain at most one effect variable. We also extend the syntax with effect masking $E\backslash L$, which means the effect types given by masking L from E. The latter is needed to keep the syntax of effect contexts closed under the masking operation $E - L$; otherwise we cannot define $\varepsilon - L$. In other words, the syntax of effects is the free algebra generated from extending D, E and masking $E\backslash L$ with base elements $\cdot$ and ε.

The effect equivalence and subeffecting rules are extended in a relatively standard way.

$$\frac{}{E\backslash\cdot \equiv E} \qquad \frac{}{\cdot\backslash L \equiv \cdot} \qquad \frac{}{(\ell : P, E)\backslash(\ell, L) \equiv E\backslash L} \qquad \frac{\ell \notin L}{(\ell : P, E)\backslash L \equiv \ell : P, E\backslash L}$$

$$\frac{}{(\varepsilon\backslash L)\backslash L' \equiv \varepsilon\backslash(L, L')} \qquad \frac{}{\varepsilon\backslash L \equiv \varepsilon\backslash L} \qquad \frac{}{\cdot \leqslant \varepsilon\backslash L} \qquad \frac{}{\varepsilon\backslash L \leqslant \varepsilon\backslash L}$$

We do not allow non-trivial equivalence or subtyping between different effect variables. We always identify effects up to the equivalence relation. That is, we can directly treat syntax of effects as the free algebra quotiented by the equivalence relation $E \equiv F$. Observe that using the equivalence relation, all open effect types with effect variable ε can be simplified to an equivalent normal form $D, \varepsilon\backslash L$. We assume the operation $E - L$ is defined for effects E in normal form and extend it with one case for effect variables as $\varepsilon\backslash L - L' = \varepsilon\backslash(L, L')$.

7 Simple Bidirectional Type Checking and Elaboration

In this section we outline the design of Metl, a basic surface language on top of Met, which uses a simple bidirectional typing strategy to infer all boxing and unboxing [16, 42].

The bidirectional typing rules for simply-typed λ-calculus and modalities of Metl and its type-directed elaboration to Met are shown in Figure 4. We show the full typing and elaboration rules and discuss our implementation in Appendix D.

$\boxed{\Gamma \vdash M \Rightarrow A \mathbin{@} E \dashrightarrow N}$ $\boxed{\Gamma \vdash M \Leftarrow A \mathbin{@} E \dashrightarrow N}$

B-Var
$$\frac{\nu_F = \mathsf{locks}(\Gamma') : E \to F \qquad \overline{\zeta}G = \mathsf{across}(\Gamma; A; \nu; F)}{\Gamma, x : \overline{\mu}G, \Gamma' \vdash x \Rightarrow \overline{\zeta}G \mathbin{@} E \dashrightarrow \mathbf{mod}_{\overline{\zeta}}\, x}$$

B-Mod
$$\frac{\mu_F : E \to F \qquad \Gamma, \blacksquare_{\mu_F} \vdash V \Leftarrow A \mathbin{@} E \dashrightarrow V'}{\Gamma \vdash V \Leftarrow \mu A \mathbin{@} F \dashrightarrow \mathbf{mod}_{\mu}\, V'}$$

B-Abs
$$\frac{\Gamma, x : \overline{\mu}G \vdash M \Leftarrow B \mathbin{@} E \dashrightarrow M'}{\Gamma \vdash \lambda x.M \Leftarrow \overline{\mu}G \to B \mathbin{@} E \dashrightarrow \lambda x.\mathbf{let}\ \mathbf{mod}_{\overline{\mu}}\, \hat{x} = x\ \mathbf{in}\ M'}$$

B-App
$$\frac{\Gamma \vdash M \Rightarrow \overline{\mu}(A \to B) \mathbin{@} E \dashrightarrow M' \qquad \overline{\mu}_E \Rightarrow \langle\rangle_E \qquad \Gamma \vdash N \Leftarrow A \mathbin{@} E \dashrightarrow N'}{\Gamma \vdash M\ N \Rightarrow B \mathbin{@} E \dashrightarrow (\mathbf{let}\ \mathbf{mod}_{\overline{\mu}}\, x = M'\ \mathbf{in}\ x)\ N'}$$

B-Annotation
$$\frac{\Gamma \vdash V \Leftarrow A \mathbin{@} E \dashrightarrow V'}{\Gamma \vdash V : A \Rightarrow A \mathbin{@} E \dashrightarrow V'}$$

B-Switch
$$\frac{\Gamma \vdash M \Rightarrow \overline{\mu}G \mathbin{@} E \dashrightarrow M' \qquad \Gamma \vdash (\overline{\mu}, G) \Rightarrow \overline{\nu} \mathbin{@} E}{\Gamma \vdash M \Leftarrow \overline{\nu}G \mathbin{@} E \dashrightarrow \mathbf{let}\ \mathbf{mod}_{\overline{\mu}}\, x = M'\ \mathbf{in}\ \mathbf{mod}_{\overline{\nu}}\, x}$$

Fig. 4. Representative bidirectional typing and elaboration rules for Metl.

As with usual bidirectional typing, we have inference mode $\Gamma \vdash M \Rightarrow A \mathbin{@} E$ and checking mode $\Gamma \vdash M \Leftarrow A \mathbin{@} E$. They both additionally take the effect context E as an input. The highlighted part $\dashrightarrow N$ gives the elaborated term N in Met.

We write G for guarded types which do not have top-level modalities, and $\overline{\mu}$ for a sequence of modalities which can be empty. As a result, every type is in form $\overline{\mu}G$. We use the following syntactic sugar for boxing and unboxing modality sequences to simplify the elaboration.

$$\begin{aligned} \mathbf{mod}_{\overline{\mu}}\ V &\doteq \overline{\mathbf{mod}_{\mu}}\ V \\ \mathbf{let}\ \mathbf{mod}_{\overline{\mu}}\ x = M\ \mathbf{in}\ N &\doteq (\lambda x.\overline{\mathbf{let}\ \mathbf{mod}_{\mu}\ x = x\ \mathbf{in}}\ N)\ M \end{aligned}$$

Our bidirectional typing rules are mostly simple and standard. The most novel part is the usage of an auxiliary function across in B-Var defined as follows.

$$\mathrm{across}(\Gamma, A, \nu, F) = \begin{cases} A, & \text{if } \Gamma \vdash A : \mathrm{Abs} \\ \zeta G, & \text{otherwise, where } A = \overline{\mu}G \text{ and } \nu_F \backslash \overline{\mu}_F = \zeta_E \end{cases}$$

When A is absolute, we can always access the variable. Otherwise, in order to know how far we should unbox the modalities $\overline{\mu}$ of the variable, we define a right residual operation $\nu_F \backslash \mu_F$ for the modality transformation relation. Given $\mu_F : E \to F$ and $\nu_F : F' \to F$, the partial operation $\nu_F \backslash \mu_F$ fails if there does not exist $\zeta_{F'}$ such that $\mu_F \Rightarrow \nu_F \circ \zeta_{F'}$. Otherwise, it gives an indexed modality such that $\mu_F \Rightarrow \nu_F \circ (\nu_F \backslash \mu_F)$ and for any $\zeta_{F'}$ with $\mu_F \Rightarrow \nu_F \circ \zeta_{F'}$, we have $\nu_F \backslash \mu_F \Rightarrow \zeta_{F'}$. Intuitively, $\nu_F \backslash \mu_F$ gives the best solution $\zeta_{F'}$ for the transformation $\mu_F \Rightarrow \nu_F \circ \zeta_{F'}$ to hold. The concrete definition of $\nu_F \backslash \mu_F$ is given in Appendix D.1.

B-App unboxes M when it has top-level modalities and inserts explicit unboxing in the elaborated term. B-Mod introduces a lock into the context and inserts explicit boxing in the elaborated term. B-Annotation is standard for bidirectional typing. B-Switch not only switches the direction from checking to inference, but also transforms the top-level modalities when there is a mismatch by inserting unboxing and re-boxing. It uses the judgement $\Gamma \vdash (\mu, A) \Rightarrow \nu \mathbin{@} E$ defined in Section 4.4.

Though incorporating polymorphic type inference is beyond the scope of this paper, we are confident that modal effect types are compatible with it. The key observation here is that in the presence of polymorphism, the problem of automatically boxing and unboxing is closely related to that of inferring first-class polymorphism. Modality introduction is analogous to type abstraction (which type inference algorithms realise through generalisation). Modality elimination is analogous to type application (which type inference algorithms realise through instantiation). As such, one can adapt any of the myriad techniques for combining first-class polymorphism with Hindley-Milner type inference. As we adopt a bidirectional type system, we could simply follow the literature on extending bidirectional typing with sound and complete inference for higher-rank polymorphism [17], first-class polymorphism [60] and bounded quantification [14].

In the future, we plan to further explore type inference for modal effect types and in particular design an extension to OCaml, building on and complementing recent work on modal types for OCaml [34] and making use of existing techniques for supporting first-class polymorphism.

8 Related and Future Work

8.1 Capability-Based Effect Systems

Capability-based effect systems such as Effekt [8, 9] and $\mathrm{CC}_{<:\Box}$ [7] interpret effects as capabilities and offer a form of effect polymorphism through capability passing.

For instance, in Effekt the `asList` handler in Section 2.3 has the following type:

```
def asList{ f: Unit ⇒ List[Int] / { Gen[Int] } }: List[Int] / {}
```

The special *block parameter* (or *capability*) `f` can use the effect `Gen[Int]` in addition to those from the context. The annotation `{ Gen[Int] }` is similar to our relative modalities.

A key difference between Effekt and Met is that Effekt requires blocks to be second-class, whereas Met supports first-class functions by default. For instance, consider the standard composition function: `compose f g x = g (f x)`. We cannot write this function naively in Effekt as it relies on first-class functions. One solution is to uncurry it.

```
def composeUncurried[A, B, C](x: A){ f: A ⇒ B }{ g: B ⇒ C }: C / {}
```

Note that we move `x` to be the first argument, as Effekt requires value parameters to appear before block parameters. We cannot partially apply `composeUncurried`.

Brachthäuser et al. [8] recover first-class functions in Effekt by boxing blocks. However, such boxed blocks can only use those capabilities specified in the box types, similarly to our absolute modalities. With boxes, we can write the curried `compose` with the following type signature

```
def compose[A, B, C]{ f: A ⇒ B }{ g: B ⇒ C }: A ⇒ C at {f, g} / {}
```

which returns a value of type `A ⇒ C at {f, g}` — a first-class boxed block. The annotation `{f, g}` indicates that this block captures the capabilities `f` and `g`. This kind of annotation is reminiscent of effect variables, and indeed such examples illustrate why Met without effect polymorphism is not expressive enough to encode all of Effekt. If we extend Met with effect variables (Section 6.2) then it is possible to encode capability variables like `f` and `g`.

Another key difference between Met and Effekt is that Effekt uses named handlers [6, 55, 59], in which operations are dispatched to a specific named handler, whereas Met uses Plotkin and Pretnar [43]-style handlers, in which operations are dispatched to the first matching handler in the dynamic context. Named handlers also provide a form of effect generativity. In future it would be interesting to explore variants of modal effect types with named handlers and generative effects [15].

$CC_{<:\square}$ [7], the basis for capture tracking in Scala 3, also provides succinct types for uncurried higher-order functions like `composeUncurried`. As in Effekt, the curried version requires the result function to be explicitly annotated with its capture set `{f,g}`. Though $CC_{<:\square}$ is a good fit for Scala 3, it does rely on existing advanced features like path-dependent types (especially the ability of using term variables in types) and implicit parameters. Modal effect types do not require the language to support such advanced features.

8.2 Direct Comparison between Metl, Koka, and Effekt

In Section 2, we compare the types of `iter`, `asList`, `state`, and `schedule` in Met with their counterparts in a row-based effect system similar to Koka. In Section 8.1, we discuss the differences between modal effect types and capability-based effect systems such as Effekt. Here we present a more direct comparison of the type signatures of typical programs in Metl, Koka, and Effekt, in order to demonstrate the practicality of modal effect types.

Invoking Effects. An effect system tracks which effects are invoked. Consider a function `foo x = do yield (x + 42)` which uses the `yield` operation from Section 2.2. Its types in Metl, Koka, and Effekt are as follows.

```
foo : [Gen Int](Int → 1)                    # METL
foo : forall<e>. (x : int) → <gen<int>|e> ()  # Koka
def foo(x : Int) : Unit / { Gen[Int] }      # Effekt
```

Both Metl and Effekt support effect subtyping which enables us to apply `foo` with other effects. Koka does not support effect subtyping and uses an effect variable `e` for modularity. Koka supports a special typing rule which implicitly inserts effect variables on function types in covariant position. Consequently, we can in fact simply write the following type in Koka.

```
foo : (x : int) → (gen<int>) ()                # Koka
```

Handling Effects. An effect system tracks which effects are handled. Recall the `asList` handler from Section 2.3. Its types in Metl, Koka, and Effekt are as follows.

```
asList : [](<Gen Int>(1 → 1) → List Int)                              # METL
asList : forall<e>. (action : () → <gen<int>|e> ()) → e list<int>  # Koka
def asList{ f: Unit ⇒ List[Int] / { Gen[Int] } }: List[Int] / {}    # Effekt
```

Both Metl and Effekt allow the argument of `asList` to use the ambient effects in addition to `Gen Int`. In Koka, we must make the argument polymorphic over other effects.

Functions in Data Types. An effect system should be compatible with algebraic data types. Consider a pair of `foo` functions as defined in Section 8.2.

In Metl, we can choose to either share a single absolute modality between both components, or to have a separate absolute modality for each component.

```
pair1 : [Gen Int](Int → 1, Int → 1)
pair2 : ([Gen Int](Int → 1), [Gen Int](Int → 1))
```

The values `pair1` and `pair2` can be easily converted between one another.

In Koka, we must make both functions effect-polymorphic. We can choose either to use the same or different effect variables:

```
pair3 : forall<e,e1>. ((x : int) → <gen<int>|e> (), (y : int) → <gen<int>|e1> ())
pair4 : forall<e>.    ((x : int) → <gen<int>|e> (), (y : int) → <gen<int>|e> ())
```

In Effekt, as discussed in Section 8.1, we cannot express this pair of functions directly, as functions are second-class. We must box the functions to make them first-class.

```
pair5 : Tuple2[Int ⇒ Unit / { Gen[Int] } at {}, Int ⇒ Unit / { Gen[Int] } at {}]
```

The syntax `at {}` means that preceding function type is boxed with no captured capability.

Higher-Order Functions. An effect system should be compatible with higher-order functions. Consider the standard sequential composition function `compose f g x = g (f x)`. Its type in Metl and Koka are as follows.

```
compose : forall a b c . []((a → b) → (b → c) → (a → c))                  # METL
compose : forall<a,b,c,e>. (f : (a) →e b) → (g : (b) →e c) → ((x : a) →e c)# Koka
```

In Effekt, as we discussed in Section 8.1, we must either switch to an uncurried version or box the result with capability variables.

```
def composeUncurried[A, B, C](x: A){ f: A ⇒ B }{ g: B ⇒ C }: C / {}  # Effekt
def compose[A, B, C]{ f: A ⇒ B }{ g: B ⇒ C }: A ⇒ C at {f, g} / {}  # Effekt
```

In summary, compared to Koka, Metl provides more succinct types without effect variables for a rich class of programs. Compared to Effekt, Metl supports first-class functions smoothly with no extra restrictions or capability variables in types.

8.3 Frank

Our absolute and relative modalities are inspired by *abilities* and *adjustments* in Frank [13, 33]. Absolute modalities and abilities specify the whole effect context required to run some computation. Relative modalities and adjustments specify changes to the ambient effect context. The key difference is that Frank is based on a traditional row-based effect system and implicitly inserts effect variables into higher-order programs. This is a fragile syntactic abstraction as discussed in Section 1. In contrast, Met exploits modal types to robustly capture the essence of modular effect programming

without effect polymorphism. As demonstrated in Section 5, a core Frank-like calculus with implicit effect variables is expressible in Met. Frank's *adaptors* are richer than Met's masking, though we expect relative modalities to extend readily to encompass the full power of adaptors.

Unlike adjustments in Frank, modal types are first-class types just like data types and can appear anywhere. For instance, we can put two functions with modal types in a pair.

```
handleTwo : [](([Gen Int](1 → 1), [State Int](1 → 1)) → (List Int, 1))
handleTwo (f, g) = (asList f, state g 42)
```

8.4 Relationship Between Met and Multimodal Type Theory

The literature on multimodal type theory organises the structure of modes (objects), modalities (morphisms between objects), and their transformations (2-cells between morphisms) in a *2-category* [18, 19, 30] (or, in the case of a single mode, a semiring [1, 10, 39, 40]). In Met, modes are effect contexts E, modalities are of the form $\mu_F : E \to F$, and transformations are of the form $\mu_F \Rightarrow \nu_F$. However, 2-categories are insufficient in a system that also includes submoding. The extra structure can be captured by moving to *double categories*, which have an additional kind of vertical morphism between objects (in Met, vertical morphisms are given by the subeffecting relation $E \leqslant F$), as also proposed by Katsumata [29]. Consequently, the transformations do not strictly require the two modalities to have the same sources and targets, enabling us to have $[]_F \Rightarrow [E]_F$ in Met. The relationship between Met and MTT is explained in more detail in Appendix B.1.

8.5 Other Related Work

We discuss other related work on effect systems and modal types.

Row-based Effect Systems. Row polymorphism is one popular approach to implementing effect systems for effect handlers. Links [21] uses Rémy-style row polymorphism with presence types [45], whereas Koka [32] and Frank [33] use scoped rows [31] which allow duplicated labels. Morris and McKinna [36] propose a general framework for comparing different styles of row types, and Yoshioka et al. [58] propose a similar framework focusing on comparing effect rows. Met adopts Leijen-style scoped rows, but also allows operation labels to be absent in the spirit of Rémy-style presence types.

Subtyping-based Effect Systems. Eff [3, 44] is equipped with an effect system with both effect variables and sub-effecting based on the type inference and elaboration described in Karachalias et al. [28]. The effect system of Helium [6] is based on finite sets, offering a natural sub-effecting relation corresponding to set-inclusion. As such, their system aligns closely with Lucassen and Gifford [35]-style effect systems. Tang et al. [50] propose a calculus for effect handlers with effect polymorphism and sub-effecting via qualified types [26, 36].

Modal Types and Effects. Choudhury and Krishnaswami [11] propose to use the necessity modality to recover purity from an effectful calculus, which is similar to our empty absolute modality. Zyuzin and Nanevski [61] extend contextual modal types [37] to algebraic effects and handlers. Their system lacks anything like our relative modality and thus cannot benefit from ambient effect contexts due to strict syntactic restrictions. Consequently, they cannot provide concise modular types for higher-order functions and handlers as Met does.

8.6 Future Work

Future work includes: implementing our system as an extension to OCaml; exploring extensions of modal effect types with Fitch-style unboxing, named handlers, and capabilities; combining modal effect types with control-flow linearity [50]; and developing a denotational semantics.

Acknowledgments

Sam Lindley was supported by UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (MR/T043830/1 and MR/Z000351/1).

Data-Availability Statement

The prototype of our surface language Metl is available on Zenodo [52]. This prototype implements and extends the bidirectional type checker outlined in Section 7 and appendix D. It type checks all examples in the paper.

References

[1] Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. doi:10.1145/3408972

[2] Danel Ahman. 2023. When Programs Have to Watch Paint Dry. In *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 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. 13992)*, Orna Kupferman and Pawel Sobocinski (Eds.). Springer, 1–23. doi:10.1007/978-3-031-30829-1_1

[3] Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123.

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (jan 2021), 32 pages. doi:10.1145/3434331

[11] 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

[12] Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 258–275. doi:10.1007/978-3-319-89366-2_14

[13] 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

[14] Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2083–2111. doi:10.1145/3622871

[15] 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

[16] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. doi:10.1145/3450952

[17] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. doi:10.1145/2500365.2500582

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

[19] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 492–506. doi:10.1145/3373718.3394736

[20] 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

[21] 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

[22] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe*. ACM, 15–27.

[23] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 415–435. doi:10.1007/978-3-030-02768-1_22

[24] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. doi:10.1017/S0956796820000040

[25] Daniel Hillerström. 2022. *Foundations for Programming and Implementing Effect Handlers*. Ph. D. Dissertation. The University of Edinburgh, UK. doi:10.7488/era/2122

[26] Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994), 231–256. doi:10.1016/0167-6423(94)00005-0

[27] 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

[28] Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. 2020. Explicit effect subtyping. *J. Funct. Program.* 30 (2020), e15. doi:10.1017/S0956796820000131

[29] Shin-ya Katsumata. 2018. A Double Category Theoretic Analysis of Graded Linear Exponential Comonads. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 110–127.

[30] 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

[31] 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.

[32] 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

[33] 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

[34] 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). https://antonlorenzen.de/oxidizing-ocaml-modal-memory-management.pdf

[35] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57. doi:10.1145/73560.73564

[36] 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

[37] 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

[38] Max S. New, Eric Giovannini, and Daniel R. Licata. 2023. Gradual Typing for Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1758–1786. doi:10.1145/3622860

[39] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (jul 2019), 30 pages. doi:10.1145/3341714

[40] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. doi:10.1145/2628136.2628160

[41] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. doi:10.1145/3622814

[42] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. doi:10.1145/345099.345100

[43] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).
[44] Matija Pretnar. 2014. Inferring Algebraic Effects. *Log. Methods Comput. Sci.* 10, 3 (2014). doi:10.2168/LMCS-10(3:21)2014
[45] 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.
[46] Dennis Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* 17, 7 (1974), 365–375.
[47] Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Math. Struct. Comput. Sci.* 28, 6 (2018), 856–941. doi:10.1017/S0960129517000147
[48] Michael Shulman. 2023. Semantics of multimodal adjoint type theory. In *Proceedings of the 39th Conference on the Mathematical Foundations of Programming Semantics, MFPS XXXIX, Indiana University, Bloomington, IN, USA, June 21-23, 2023 (EPTICS, Vol. 3)*, Marie Kerjean and Paul Blain Levy (Eds.). EpiSciences. doi:10.46298/ENTICS.12300
[49] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. doi:10.1145/3453483.3454039
[50] Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL, Article 54 (jan 2024), 29 pages. doi:10.1145/3632896
[51] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *CoRR* abs/2407.11816 (2025). doi:10.48550/ARXIV.2407.11816
[52] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. *OOPSLA25 Artifact for Modal Effect Types*. doi:10.5281/zenodo.14982742
[53] Birthe van den Berg and Tom Schrijvers. 2023. A Framework for Higher-Order Effects & Handlers. *CoRR* abs/2302.01415 (2023). doi:10.48550/arXiv.2302.01415 arXiv:2302.01415
[54] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *SIGPLAN Not.* 49, 12 (Sept. 2014), 1–12. doi:10.1145/2775050.2633358
[55] 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
[56] Xu Xue and Bruno C. d. S. Oliveira. 2024. Contextual Typing. *Proc. ACM Program. Lang.* 8, ICFP, Article 266 (Aug. 2024), 29 pages. doi:10.1145/3674655
[57] Zhixuan Yang and Nicolas Wu. 2023. Modular Models of Monoids with Operations. *Proc. ACM Program. Lang.* 7, ICFP (2023), 566–603. doi:10.1145/3607850
[58] 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
[59] 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
[60] Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. doi:10.4230/LIPICS.ECOOP.2022.2
[61] 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 2024-10-16; accepted 2025-02-18