Higher-order fork, modally

Aghilas Y. Boussaa
aghilas.boussaa@ens.fr
École normale supérieure – PSL
France

Wenhao Tang
wenhao.tang@ed.ac.uk
The University of Edinburgh
UK

Sam Lindley
sam.lindley@ed.ac.uk
The University of Edinburgh
UK

Abstract

Effect handlers are a powerful programming construct allowing users to define and combine computational effects, such as cooperative concurrency. Effect systems statically track the effects that a program can perform. Traditional effect systems rely on effect variables to provide modular type signatures for higher-order functions. Recent work on modal effect types (Met) reduces, and often eliminates altogether, the need for effect variables via modalities. Alas, the standard way to define a higher-order fork operation whose child process is parametric in its effects is to extend Met with effect variables. We propose a small extension to Met, based on introducing an ordering on effects in effect contexts, that enables us to define a higher-order fork operation whose child process is parametric in its effects, without the need for any effect variables at all.

1 Introduction

Effect handlers can express many control-flow constructs. One example is cooperative concurrency [8], which can be modelled using an effect with two operations: fork creates a new child process; yield pauses the current process. There are two canonical forms of fork operation we might consider.

The first form of fork is a UNIX(-style) fork operation of type unit ⇒ bool [5] (we denote operation signatures with a fat arrow). The continuation receives a boolean for distinguishing the parent and child in the following code.

```
if do fork () then print "parent" else print "child"
```

UNIX fork relies on multi-shot continuations, so is unsuitable for programming languages that support only single-shot continuations [10, 9].

The second form of fork is a higher-order fork operation of type (unit → unit) ⇒ unit [4, 10]. This operation takes a thunk for the child process, which is a function of type unit → unit. The program corresponding to our previous example is written as follows with higher-order fork.

```
do fork (fun () → print "child");
print "parent"
```

Higher-order fork can be implemented using single-shot continuations. However, higher-order fork is somewhat complicated by the need to keep track of the effects in its parameter. Traditional effect systems annotate function arrows with effects, so the argument type (unit → unit) of fork would typically be annotated with the effects that the child process might perform. Annotating arrows with the effects they can perform between square brackets, a suitable scheduler could have a type along the lines:

```
forall e . (unit →[co e, queue (proc e), e] unit)
  →[co e, queue (proc e), e] unit
```

where `co` is the concurrency effect (admitting `fork` and `yield` operations), the `queue` effect abstracts over the queue of concurrent processes and `proc` is a type of suspended processes. We rely on the effect variable `e` to parameterise over the effects that the child process may perform.

Modal effect types (Met) offer elegant means of avoiding effect variables for higher-order functions, but still require effect variables for higher-order operations such as higher-order fork. We write Met$^{\epsilon}$ for the extension of Met with effect variables. In the rest of this extended abstract, we briefly describe implementations of higher-order fork in existing effect systems (Koka, Frank, Effekt, and Met$^{\epsilon}$), demonstrate why effect variables are needed, and introduce Met$^{\sharp}$, a small extension to Met which supports higher-order fork without effect variables.

2 Higher-order fork in existing languages

In this section, we briefly sketch how to implement a cooperative concurrency effect incorporating a higher-order fork operation in existing languages with effect handlers and effect systems. Our implementations of schedulers use a parametric queue effect to store thunks. The full implementations are available in the appendix and online.

Koka. Koka has a traditional type-and-effect system where each arrow type holds a scoped row of the effects the function can perform [7, 6]. We use an effect variable for the fork operation and to define an effect-polymorphic type of processes. The same effect variable appears on all arrows of the scheduler's type.

Frank. Embodying the observation that effect variables are almost always the same throughout a signature, the Frank programming language omits effect variables by default and uses syntactic sugar to automatically insert them [3]. This allows implementing concurrency with a higher-order fork operation without any explicit effect variables [3, Section 6]. However, effect variables still exist during type checking and may leak into error messages displayed to the programmer.

Effekt. Effekt tracks effects as capabilities [1, 2]. Functions in Effekt are second-class citizens by default. In order to make them first-class so that we can use them as arguments of

fork, we have to box them and specify the capabilities they use explicitly. In order to parameterise over the capabilities that the child process might use, we must introduce an explicit capability variable, which is reminiscent of the effect variables in Koka and Met$^{\epsilon}$.

3 Higher-order fork in Met$^{\epsilon}$

Modal effect types (Met) decouple effects from function types via modalities [12, 11] and enable type signatures similarly succinct to Frank's without relying on fragile syntactic sugar. The central notions of Met are the *effect context*, which determines the effects provided by the environment, and *modalities*, which transform the effect context. For example, a `map` function has the following type signature.

```
val map : forall a b. []((a → b) → list a → list b)
```

The `[]` is an *absolute modality* which transforms the ambient effect context by discarding it and replacing it with the empty effect context, since the definition of `map` itself uses no effect. We can apply `map` to any effectful function, as long as the effect context at the call site provides the required effects. We refer to [12, 11] for a full introduction to Met.

Met enables a large fragment of effect polymorphic programs in Met to be written without using effect variables [12]. However, this fragment does not include programs that use higher-order effects. To ensure effect safety, operations can only yield values whose meaning is independent of the ambient effect context, which is not the case for arrow types, as they may use any effect in the context. Yielding a function is only possible by first boxing it in an *absolute* modality that fixes the set of effects it can perform. To be able to compose cooperative concurrency with any effect, we cannot fix a set of effects and rely on an extension with effect variables [12, Section 2.10].

```
eff co (e : Effect)
  = fork : [co e, e](unit → unit) ⇒ unit
  | yield : unit ⇒ unit
```

We define a recursive polymorphic type of processes.

```
type proc (e : Effect) =
  Proc of [queue (proc e), e](unit → unit)
```

Then, the scheduler has the following type.

```
forall (e : Effect) . [queue (proc e), e]
  ([co e, queue (proc e), e](unit → unit) → unit)
```

Because each arrow has an absolute modality, this signature is similar to the one in the introduction, and loses the advantages of modal effect types. Moreover, this implementation relies on a *modality-parameterised handler* [11, Section 3.5]. These handlers wrap the continuation with a user-supplied modality. Here, this allows us to fix the effects the continuation can perform with an absolute modality.

4 Higher-order fork in Met$^{\sharp}$

As we have seen, in order to implement a higher-order fork effect that allows the child process to perform any effect, we have to use effect variables either explicitly or implicitly in existing effect systems. This is particularly unsatisfactory for Met, as Met is designed to avoid effect variables.

We observe that the main challenge is that when calling a higher-order fork in Met, we do not know what the effect context for the argument (the child process) should be, as it depends on the effect context of the handler of fork. As a result, we must fully specify which effects may be performed by the child process via an absolute modality as in Section 3, and consequently, we use effect variables to parameterise over the effects that the child process might perform.

We propose Met$^{\sharp}$, a simple solution to avoid effect variables. Our key idea is to impose an order on effects in effect contexts reflecting the order of handlers in the evaluation context. When we call a higher-order operation such as fork, the effects available when the child process is called are exactly those that follow the co effect in the effect context. The cooperative concurrency effect is now defined as follows.

```
eff# co = fork : (unit → unit) ⇒ unit
        | yield : unit ⇒ unit
```

The hash (#) indicates that co is a higher-order effect, allowing fork to take as input a function. For example, in presence of a function `f : <queue unit>(unit → unit) → unit`, the typing rules reject the unsafe program below, which would allow the queue effect to leak [12, Section 2.10].

```
handle f (fun () -> do fork (fun () -> do enqueue ())) with
  | fork p _ ⇒ p -- we omit the other clauses
```

The modality `<queue unit>` is a *relative* modality, which extends the ambient effect context with `queue unit`. As a result, the effect context for the underlined thunk would be `queue unit, co`. From this ordering, Met$^{\sharp}$ knows that the queue effect would not be available when the child process is called at the handler of fork, rejecting this program.

The signature of fork does not mention the co effect, because the thunk will have the same ambient effect context as the computation that yielded it, which contains effect co. We can further eliminate the `proc` datatype and just use plain functions. The scheduler does not need a modality-parameterised handler and has the following type.

```
[queue (unit → unit)](<co>(unit → unit) → unit)
```

In summary, Met$^{\sharp}$ offers a structural way to determine the set of effects the operands an operation can perform. In the case of higher-order fork, this eliminates the need for effect variables entirely. Future work includes: extending modalities to also act on effects to increase the flexibility of Met$^{\sharp}$, and inserting masks instead of requiring higher-order operations to come from the first effect in the context.

References

[1] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2020. DOI: 10.1145/3428194.

[2] Jonathan Immanuel Brachthäuser et al. "Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2022. DOI: 10.1145/3527320.

[3] Lukas Convent et al. "Doo Bee Doo Bee Doo". In: *Booktitle of Functional Programming (JFP)* 30 (2020), e9. DOI: 10.1017/S0956796820000039.

[4] Stephen Dolan et al. "Effective concurrency through algebraic effects". In: *OCaml Workshop*. 2015. URL: https://www.cl.cam.ac.uk/~jdy22/papers/effective-concurrency-through-algebraic-effects.pdf.

[5] Daniel Hillerström. "Foundations for Programming and Implementing Effect Handlers". PhD thesis. School of Informatics, The University of Edinburgh, 2021. DOI: 10.7488/era/2122.

[6] Daan Leijen. "Extensible records with scoped labels". In: *Symposium on Trends in Functional Programming (TFP)*. 2005. URL: https://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/21num.pdf.

[7] Daan Leijen. "Koka: Programming with row polymorphic effect types". In: *Electronic Proceedings in Theoretical Computer Science (EPTCS)*. 2014. DOI: 10.4204/EPTCS.153.8.

[8] Daan Leijen. "Structured asynchrony with algebraic effects". In: *Workshop on Type-Driven Development (TyDe)*. 2017. DOI: 10.1145/3122975.3122977.

[9] Luna Phipps-Costin et al. "Continuing WebAssembly with effect handlers". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2023. DOI: 10.1145/3622814.

[10] KC Sivaramakrishnan et al. "Retrofitting effect handlers onto OCaml". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2021. DOI: 10.1145/3453483.3454039.

[11] Wenhao Tang and Sam Lindley. "Rows and Capabilities as Modal Effects". In: *Symposium on Principles of Programming Languages (POPL)*. 2026. DOI: 10.1145/3776674.

[12] Wenhao Tang et al. "Modal effect types". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2025. DOI: 10.1145/3776674.

A Koka

```
div effect co<e>
  ctl yield () : unit
  ctl fork (p : () → <co<e>,div|e> unit) : unit

effect queue<a>
  ctl enqueue (x : a) : unit
  ctl dequeue () : maybe<a>

div type proc<e>
  Proc (p :() → <queue<proc<e>>,div|e> unit)

fun run_next() : <queue<proc<e>>, div|e> unit
  match dequeue()
    Nothing → ()
    Just (Proc (p)) → p()

fun schedule(p : () → <co<e>, queue<proc<e>>, div|e> unit)
    : <queue<proc<e>>,div|e> unit
  with handler
    return (_)
      run_next()
    ctl yield ()
      enqueue (Proc ({ resume (()) }))
      run_next()
    ctl fork(p')
      enqueue (Proc ({ schedule({ mask<queue> (p') })}))
      resume(())
  p ()
```

B Frank

```
data Maybe X = just X | nothing

interface Co = fork : {[Co]Unit} → Unit
             | yield : Unit

interface Queue S = enqueue : S → Unit
                  | dequeue : Maybe S

data Proc = proc {[Queue Proc]Unit}

runNext : {[Queue Proc]Unit}
runNext! = case dequeue! { (just (proc x)) → x!
                         | nothing         → unit }

schedule : {<Co>Unit → [Queue Proc]Unit}
schedule <yield → k>  =
  enqueue (proc {schedule (k unit)});
  runNext!
schedule <fork p → k> =
  enqueue (proc {schedule (<Queue> p!)});
  schedule (k unit)
schedule unit = runNext!
```

C Met$^{\epsilon}$

```
type unit = Unit

type option a = None | Some of a

eff co (e : Effect)
  = fork : [co e, e]{unit} ⇒ unit
  | yield : unit ⇒ unit

eff queue [s]
  = enqueue : s ⇒ unit
  | dequeue : unit ⇒ option s

type proc (e : Effect) = Proc of [queue (proc e), e]{unit}

val wakeProc : forall (e : Effect) . [queue (proc e), e]{unit}
let wakeProc! =
  match do dequeue () with
  | Some (Proc (x)) → x!
  | None → ()
  end

val schedule : forall (e : Effect) .
```

```
  [queue (proc e), e]([co e, queue (proc e), e]{unit} → unit)
let schedule c =
  handle<co e>[queue (proc e), e] c! with
  | return () ⇒ (wakeProc @e)!
  | yield () k ⇒
    do enqueue (Proc (k));
    (wakeProc @e)!
  | fork p k ⇒
    do enqueue (Proc ({schedule @e {mask<queue> (p!)}}));
    k!
  end
```

D Effekt

D.1 Boxed functions

```
import dequeue

interface Co[A, B] {
  def yield(): Unit
  def fork(p: Process[A, B]): Unit
}

type Process[A, B] =
  ({ cap: A ⇒ B } ⇒ (() ⇒ Unit / Co[A, B] at {cap})) at {}

def scheduler[A, B](prog: Process[A, B]) { cap: A ⇒ B } =
  region this {
  var queue: Dequeue[() ⇒ Unit at {this, cap}] in this =
    emptyQueue();

  def run_next(): Unit = queue.popBack match {
    case None() ⇒ ()
    case Some((p, q)) ⇒
      queue = q
      p()
  }

  def schedule(proc: () ⇒ Unit / Co[A, B] at {cap}): Unit = {
    try { proc() } with Co[A, B] {
      def yield() = {
        queue = queue.pushFront(box { resume(()) });
        run_next()
      }
      def fork(child) = {
        queue = queue.pushFront(box { resume(()) });
        schedule(unbox child { cap })
      }
    };
    run_next()
  }

  schedule(unbox prog { cap })
}
```

D.2 Bidirectional effects

Effekt supports bidirectional effects which can be used to implement some form of higher-order effects. With bidirectional effects, fork can take as input second-class *blocks*.

```
  interface Co {
    def yield(): Unit
    def abort(): Nothing
    def forkOp{ p: () ⇒ Unit }: Unit }
```

When handling `forkOp`, the continuation expects as argument a function that takes `p` as input. The scheduler resumes the continuation twice: to enqueue `p` and to continue the computation. To continue the parent computation only once, the `fork` function wraps each fork with an `abort` operation.

```
  def fork { p: ⇒ Unit / Co }: Unit / Co =
    do forkOp { p(); do abort() }
```

The full implementation is as follows.

```
import dequeue

interface Co {
  def yield(): Unit
  def abort(): Nothing
  def forkOp { p: () ⇒ Unit }: Unit
}

def fork { p: ⇒ Unit / Co }: Unit / Co =
  do forkOp {
    p();
    do abort()
  }

def scheduler { prog: ⇒ Unit / Co } = region this {
  var queue: Dequeue[() ⇒ Unit at {this, prog}] in this =
    emptyQueue();

  def run_next(): Unit = queue.popBack match {
    case None() ⇒ ()
    case Some((p, q)) ⇒
      queue = q
      p()
  }

  def schedule(): Unit = {
    try { prog() } with Co {
      def yield() = {
        queue = queue.pushFront(box { resume(()) });
        run_next()
      }
      def forkOp() = {
        queue = queue.pushFront(box {
          resume { {child: () ⇒ Unit} ⇒ child() }
        });
        resume { {child: () ⇒ Unit} ⇒
          ()
        }
      }
      def abort() = ()
    };
    run_next()
  }

  schedule()
```

```
}
```

D.3 Additional comments

The version with bidirectional effects resumes the continuation twice. Effekt's type system reject the following handler for `forkOp`, which would only require resuming once, because the capability `child` is incompatible with the queue.

```
def forkOp() =
  resume { {child: () ⇒ Unit} ⇒
    queue.pushFront(box child);
    () }
```

Such a handler, would remove the need for the wrapper.

We have not managed to use a queue effect to store our thunks. To do so, we would must store in a queue a thunk that captures the queue capability itself. However, we could not implement such a queue while staying in Effekt's lexical requirements.

E MET♯

```
type unit = Unit

type option a = None | Some of a

eff# co = fork : (unit → unit) ⇒ unit
        | yield : unit ⇒ unit

eff# queue a = enqueue : a ⇒ unit
             | dequeue : unit ⇒ option a

val next : [queue {unit}](unit → unit)
let next () =
  match do dequeue () with
  | None → ()
  | Some (p) → p!
  end

val schedule : [queue {unit}](<co>{unit} → unit)
let schedule m =
  handle m! with
  | return _ ⇒ next!
  | yield _ k ⇒ do enqueue k; next!
  | fork p k ⇒ do enqueue { schedule p }; k!
  end

val full_schedule : [](<co>{unit} → unit)
let full_schedule m =
  queue { schedule { mask<queue> (m!) } }
```

E.1 Theory

We here spell out the changes to the formal rules of MET($\mathcal{R}_{\text{scp}}$) [11] necessary to support our extension. We only consider here a simplified version without local labels and modality-parameterised handlers. The syntax annotates each effect type with an *accidental*: a flat signature $A \Rightarrow^{\flat} B$ has operands of absolute types, and a sharp signature $A \Rightarrow^{\sharp} B$ has no restriction. Effect contexts and extensions are scoped rows of labels that a context map to signatures.

Accidentals	Y	$::=$	$\flat \mid \sharp$
Extensions	D	$::=$	$\cdot \mid \ell, D$
Effect Contexts	E, F	$::=$	$\cdot \mid \ell, E$
Label Contexts	Σ	$::=$	$\cdot \mid \ell : A \Rightarrow^{Y} B, \Sigma$

The extension equivalence rules fix the position of sharp signatures. Effect context equivalence obeys the same rules. All of the following rules suppose the existence of a global label context Σ.

$$\frac{}{\cdot \equiv \cdot} \qquad \frac{D_1 \equiv D_2 \qquad D_2 \equiv D_3}{D_1 \equiv D_3} \qquad \frac{D_1 \equiv D_2}{\ell, D_1 \equiv \ell, D_2}$$

$$\frac{\ell \neq \ell' \qquad \ell : \flat \qquad \ell' : \flat}{\ell, \ell', D \equiv \ell', \ell, D}$$

Subeffecting of extensions allows moving sharp labels backward.

$$\frac{}{\cdot \leqslant \cdot} \qquad \frac{D_1 \leqslant D_2 \qquad D_2 \leqslant D_3}{D_1 \leqslant D_3} \qquad \frac{D_1 \leqslant D_2}{\ell, D_1 \leqslant \ell, D_2}$$

$$\frac{\ell \neq \ell' \qquad \ell : \flat}{\ell, \ell', D \leqslant \ell', \ell, D}$$

For subeffecting, effect contexts follow the same rules as extensions with the addition of $\cdot \leqslant E$.

The rules for **do** ensure that a sharp operation is available only if it appears as the first operation in the ambient effect context.

T-DoFlat

$$\frac{E = D, \ell, F \qquad \ell \notin D \qquad \Sigma \ni \ell : A \Rightarrow^{\flat} B \qquad \Gamma \vdash N : A \mathbin{@} E}{\Gamma \vdash \mathbf{do}\ \ell\ N : B \mathbin{@} E}$$

T-DoSharp

$$\frac{\Sigma \ni \ell : A \Rightarrow^{\sharp} B \qquad \Gamma \vdash N : A \mathbin{@} \ell, E}{\Gamma \vdash \mathbf{do}\ \ell\ N : B \mathbin{@} \ell, E}$$

To simplify the rules, we restrict each handler to a single operation. The rule T-HandleSharp wraps the operands of sharp operations with extension modalities.

T-HandleSharp

$$\frac{\begin{array}{c}\Gamma, \text{🔒}_{\langle \mid \ell \rangle_E} \vdash M : A \mathbin{@} \ell, F \\ \Sigma \ni \ell : A' \Rightarrow^{\sharp} B' \qquad \Gamma, x : \langle \ell \rangle A \vdash N : B \mathbin{@} F \\ \Gamma, p : \langle \ell \rangle A', r : \langle \ell \rangle B' \to B \vdash N' : B \mathbin{@} F\end{array}}{\mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N, \ell\ p\ r \mapsto N'\} : B \mathbin{@} F}$$

Handling flat operation works the same way as usual in MET.