Locality and Effect Reflection

LEO WHITE, Jane Street, UK

Programming languages increasingly support algebraic effects and handlers for writing effectful code in a principled way by extending a language's *ambient monad* – the monad describing which effects ordinary functions can perform – with additional effectful operations. However, integrating effects into type systems typically requires specialized effect systems that add complexity to the language.

We introduce *effect reflection*, an alternative presentation of algebraic effects and handlers that provides, for any algebraic signature Σ and type τ, an isomorphism:

$$W_\Sigma(\Box\,\tau) \qquad \cong \qquad y(\Sigma) \to \Box\,\tau$$

where $W_\Sigma(\Box\,\tau)$ is the free algebra of Σ at $\Box\,\tau$, $y(\Sigma)$ is a type representing how to inject terms of Σ into the current ambient monad, and $\Box$ is a comonadic *global modality* used to track which values do not depend on the ambient monad. W_Σ is sometimes called the *freer monad* of Σ. This isomorphism emerges naturally from a denotational semantics for algebraic effects in terms of presheaves over bounded Lawvere theories.

Effect reflection provides all the expressiveness of typed algebraic effects without requiring language-level support for effect systems. We demonstrate this by implementing effect reflection as a library in OxCaml [11] and showing how classic effectful programs can be written in natural direct style.

1 Reflection for algebraic effects

1.1 Algebraic effects

Algebraic effects and handlers have emerged as a powerful approach to structured programming with effects. Handlers allow the programmer to temporarily extend the *ambient monad* of their language – the monad describing which effects an ordinary function can perform – with additional effectful operations.

For example, in OCaml, one can write:

```
type 'a Effect.t +=
  | Get : int Effect.t
  | Set : int -> unit Effect.t

let handle_state (f : unit -> 'a) : 'a =
  match f () with
  | x -> fun _ -> x
  | effect Get, k -> fun s -> continue k s s
  | effect Set s', k -> fun _ -> continue k () s'
```

and for the duration of the `f ()` function call the ambient monad is extended with two additional effectful operations:

```
let get () : int = perform Get
let set (i : int) : unit = perform (Set i)
```

However, effect handlers face a fundamental tension: *untracked* effects are unsafe, while *tracked* effects require complex type system extensions.

In OCaml, the effects are untracked: nothing prevents attempting to perform an algebraic effect operation that is not supported by the current ambient monad, resulting in an `Unhandled` exception being raised.

Author's Contact Information: Leo White, Jane Street, UK.

In languages like Koka [10] the type system is extended to explicitly track the effects that a function may perform. This prevents the unsafety on an unhandled effect but adds complexity to the language.

The "effects as capabilities" approach of the Effekt language [4] points the way by elegantly tracking effects without requiring explicit effect-tracking machinery in the type system. However, it is restricted to second-class functions[1]. We aim to extend this approach to first class functions.

1.2 Locality and effect reflection

Locality captures the general idea of a value depending on some notion of "location". Our key insight is that the ambient monad can be viewed as a form of location, allowing us to reuse existing type system features for tracking locality.

We can represent locality in a type system using a comonadic *global modality* $\Box$ that tracks which values do *not* depend on their location. Values of an ordinary type τ are assumed to potentially depend on the current location, whereas values of the global type $\Box\,\tau$ are known not to.

In our setting being global means not depending on the ambient monad. For example, $A \to B$ represents computations that may use operations from the ambient monad, while $\Box(A \to B)$ contains only pure computations that ignore the ambient monad.

Based on this relationship between locality and effects, we present *effect reflection*, providing, for any algebraic signature Σ and type τ, an isomorphism:

$$W_\Sigma(\Box\,\tau) \qquad \cong \qquad y(\Sigma) \to \Box\,\tau$$

where $W_\Sigma(\Box\,\tau)$ is an inductive type representing the free algebra of Σ at $\Box\,\tau$, and $y(\Sigma)$ is a type representing how to inject terms of Σ into the current ambient monad. W_Σ is sometimes called the *freer monad* of Σ [9].

This isomorphism reveals that effectful computations can be represented as ordinary functions, enabling direct-style programming with full effect safety.

1.3 Effect reflection in OxCaml

OxCaml [11] provides type system support for tracking locality and runtime support for algebraic effects. For backwards compatibility with OCaml, an unadorned arrow type `t -> s` is treated as if the parameter and return types were actually `□t` and `□s`. To get parameters or results that are not implicitly global we write `@ local` on them, e.g. `t @ local -> s @ local`.

Using this support, we can provide effect reflection as a simple API. We represent algebraic signatures as modules with a single type constructor, providing W_Σ as `'a Term(Sigma).t` and $y(\Sigma)$ as `Handler(Sigma).t`.

For example, we can use it to write an effect handler equivalent to the OCaml one above:

```
module State = struct
  type 'a t =
    | Get : int t
    | Set : int -> unit t
end

module State_eff = Reflection(State)

let handle_state (f : Handler(State).t @ local -> 'a) (i : int) : 'a =
  let rec loop (s : int) : 'a Term(State).t @ local -> 'a  = function
```

[1] That restriction is lifted in more recent work on Effekt [3], but only by falling back to explicit effect-tracking machinery.

```
    | Return x -> x
    | Op(Get, k) -> loop s (k s)
    | Op(Set s', k) -> loop s' (k ())
  in
  loop i (State_eff.reify_local f)
```

which calls `f` and passes it a local value of type `Handler(State).t`. For the duration of the call to `f` the ambient monad is extended with two additional effectful operations, which can be used via reflection:

```
let get (h : Handler(State).t @ local) : int =
  State_eff.perform Get h

let set (h : Handler(State).t @ local) (i : int) : unit =
  State_eff.perform (Set i) h
```

The locality of the `Handler(State).t` value passed to `f` ensures that these operations can only be performed when available in the ambient monad. The type system prevents this handler value from escaping the scope of `f`, which in turn prevents `get` or `set` from being called outside that scope.

1.4 Denotational semantics

This design emerges naturally from a denotational semantics where types are interpreted as presheaves over *bounded Lawvere theories* – a categorical framework that makes the connection between locality and effects precise.

The key idea is to index the semantics by the ambient monad: rather than interpreting types as sets, we interpret them as functions from ambient monads to sets. We represent ambient monads by their corresponding algebraic theories using bounded Lawvere theories.

The key constructs of effect reflection are then defined as follows:

- The ambient monad can be defined as a monad $\mathcal{T}$ on such presheaves, which maps each algebraic theory to its free algebras.
- The global modality $\Box P$ is defined as the constant presheaf that always ignores the ambient monad and instead returns $P(\bot)$, where $\bot$ is the algebraic theory with no operations or equations.
- The semantics of our handler types $y(\Sigma)$ are given by the Yoneda embedding
- The effect reflection isomorphism becomes a simple consequence of the Yoneda lemma and some properties of the sums of algebraic theories.

1.5 Structure of the paper

We proceed as follows:

- In Section 2, we give an overview of effect reflection in practice in OxCaml.
- In Section 3, we define a simply typed calculus featuring effect reflection.
- In Section 4, we describe our denotational semantics of effect reflection.
- In Section 5, we give an implementation of effect reflection in OxCaml in terms of the untracked algebraic effects it inherits from OCaml.
- In Section 6, we discuss related work.

2 Effect reflection in practice

We can provide effect reflection as a simple library API in OxCaml. In this section we describe this API and show how to use it to write a number of typical examples of algebraic effects.

2.1 Locality in OxCaml

The effect reflection API relies on OxCaml's support for tracking locality in types, so let's start with an overview of that support.

Values in OxCaml are associated with both a type – which describes how values can be introduced and eliminated – and a *mode* – which tracks properties related to operations other than introduction and elimination. Modes are a product of a number of different axes, tracking a variety of different properties, but here we are only interested in one of them: *locality*. Ignoring the other axes, a mode is either `local` or `global`.

Locality tracks whether a value can escape the current *region*. By default, the current region ends when a function returns, so local values cannot be returned from functions.

```
let leak_local x =
  let (o @ local) = Some x in
  o
Error: o
       ^
This value escapes its region.
```

The `exclave` construct allows functions to return local values.

```
let return_local x =
  exclave
    (let (o @ local) = Some x in
     o)
```

`exclave e` can only appear in tail position within a function. It ends the function's region early and then executes the expression `e`. `e` is essentially executed within the region of the caller of the function and any local values created within it can be safely returned to that caller.

The place where modes appear within the type algebra is on arrow types. A function being `global` doesn't tell you that the values it accepts or returns should be `global` too, so instead the arrow type is labelled with two modes: one for the argument and one for the return value.

```
val return_local : 'a @ local -> 'a option @ local
```

For convenience, and backwards compatibility with OCaml, either mode can be omitted, in which case it defaults to `global`.

By default, modes are deep: a local `string list` is a local list of local strings. However, the depth that a mode applies can be controlled via *modalities*. Record fields can be annotated with the `global` modality to indicate that the contents of the field are global even when the surrounding record is local. For example:

```
type 'a gbl = { g : 'a @@ global }
```

gives a record whose only field has the `global` modality. The contents of the `g` field is always global, for example:

```
let project_global (r @ local) =
  r.g
```

gives a function of type:

```
val project_global : 'a gbl @ local -> 'a @ global
```

Variant constructor arguments can also be annotated with modalities.

Note that, whilst there is a `global` modality, there is no `local` modality. It does not make sense for a value which has the ability to leave a region to contain a value which doesn't, as when the first value leaves a region it will implicitly cause the second one to leave that region as well. This means the `global` mode is always deep.

2.2 Effect reflection interface

The types and operations of our effect reflection API are parameterized by type constructors representing the operations of an algebraic effect. In OxCaml, such higher-kinded interfaces must be built using functors, so we start with a simple module type `Op` to act as the parameter type for these functors:

```
module type Op = sig type 'a t end
```

The fundamental types of the API are the `Term` and `Handler` types.

```
module Term (O : Op) : sig
  type 'a t =
    | Return : 'a @@ global -> 'a t
    | Op : 'r O.t @@ global * ('r -> 'a t) -> 'a t
end

module Handler(O : Op) : sig
  type t
end
```

These types directly implement the types from our core isomorphism: `'a Term(Sigma).t` corresponds to $W_\Sigma(\Box\,\tau)$, while `Handler(Sigma).t` corresponds to $y(\Sigma)$

For an effect signature `O`, `'a Term(O).t` is the type of terms in the free algebra of the corresponding effect – otherwise known as the freer monad [9]. `Handler(O).t` is the type of mappings from the operations of `O` into the operations of the current ambient monad. Alternatively, `Handler(O).t` can be described as representing algebraic effect handlers that are handling the current computation.

The main piece of the API is a family of isomorphisms:

```
'a Term(O).t  ≅  Handler(O).t @ local -> 'a
```

Due to the absence of mode polymorphism in OxCaml, we expose this isomorphism as three functions: `reify` and `reify_local` which implement the left-to-right direction at the `global` and `local` modes respectively; and `perform`, which is equivalent to the right-to-left direction but makes for a more convenient primitive in practice. `perform` amounts to reflection of individual operations as opposed to full terms.

```
module Reflection (O : Op) : sig
  val reify : (Handler(O).t @ local -> 'a) -> 'a Term(O).t

  val reify_local :
    (Handler(O).t @ local -> 'a) @ local -> 'a Term(O).t @ local

  val perform : 'a O.t -> (Handler(O).t @ local -> 'a)
end
```

Using `perform` we can implement reflection of global terms as:

```
let rec reflect t h =
  match t with
  | Return x -> x
  | Op(op, k) -> reflect (k (perform op h)) h
```

with type:

```
val reflect : 'a Term(O).t -> (Handler(O).t @ local -> 'a)
```

Reflection of local terms can be defined similarly.

2.3 Example: State

As shown in the introduction we can implement an effect handler for state using this API. We start by defining the operations of the effect:

```
module State = struct
  type 'a t =
    | Get : int t
    | Set : int -> unit t
end
```

Then we instantiate the reflection isomorphism for that effect and use it to write a handler:

```
module State_eff = Reflection(State)
open Term(State)

let handle_state i f =
  let rec loop (s : int) = function
    | Return x -> x
    | Op(Get, k) -> loop s (k s)
    | Op(Set s', k) -> loop s' (k ())
  in
  loop i (State_eff.reify_local f)
```

Finally we add an alias for the handler type and some convenient wrappers for the generic effects:

```
type handler = Handler(State).t

let get h =
  State_eff.perform Get h

let set h i =
  State_eff.perform (Set i) h
```

Giving us the following interface:

```
type handler

val handle_state : int -> (handler @ local -> 'a) @ local -> 'a
val get : handler @ local -> int
val set : handler @ local -> int -> unit
```

Note that we are able to use `reify_local` to allow the function passed to `handle_state` to be local.

This shows how to implement state using effect reflection, but note that the same interface can also be implemented directly using mutable state and locality:

```
type handler = int ref

let handle_state i f = f (ref i)
let get r = !r
let set r i = r := i
```

This implementation gives the same behaviour as the one built using effect reflection. Crucially it is just as easy to reason about, which in turn means it is just as easy to reason about as the state monad. In some sense, the improved reasoning ability of the state monad over ordinary mutable state is precisely captured by the notion of locality.

This transformation from effect reflection to local mutable state works for any *tail-resumptive* effect handler.

2.4 Lightweight effect polymorphism

The following function uses state to calculate the total of a list of integers:

```
let total l =
  handle_state 0 (fun h ->
    List.iter (fun x -> set h (get h + x)) l;
    get h)
```

This relies on the locality of the function parameter of `List.iter`:

```
val iter : ('a -> unit) @ local -> 'a list -> unit
```

The `local` annotation on the parameter ensures that any function passed will not escape the current region. This allows us to pass in a closure that closes over the `local` handler `h` and use it to perform some effects.

This ability for higher-order functions like `List.iter` to be reused with different ambient monads achieves the "lightweight effect polymorphism" described by Brachthäuser et al. [4] without requiring second-class functions – locality provides the necessary restrictions while maintaining first-class status.

2.5 Example: Generators

A more realistic use of effect reflection is a *generator* effect. This effect has a single operation:

```
module Gen = struct
  type 'a t =
    | Gen : int -> unit t
end
```

Along with a handler that turns computations using `Gen` into streams of integers.

```
module Gen_eff = Reflection(Gen)
open Term(Gen)

type ints =
  | Finished
  | More of int * (unit -> ints)

let handle_gen f =
```

```
  let rec loop = function
    | Return () -> Finished
    | Op(Gen i, k) -> More(i, fun () -> loop (k ()))
  in
  loop (Gen_eff.reify f)
```

Unlike the state handler, here we must use `reify` rather than `reify_local` because our intention is that the resulting stream of integers be a global value that can escape the current region. Since it closes over the continuation `k`, we must use `reify` and require that `f` be global:

```
val handle_gen : (Handler(Gen).t @ local -> unit) -> ints
```

2.6 Example: Asynchronous I/O

Perhaps the most common use case for algebraic effects is for concurrency and asynchronous I/O. Implementing asynchronous I/O from scratch is beyond the scope of this paper, but we can build a simple version on top of an existing monadic concurrency library. Assuming we have a monadic future type `'a Deferred.t` we can write concurrent code in direct style by using an effect with a single `Await` operation that waits for a future to be completed:

```
module Await : sig
  type 'a t =
    | Await : 'a Deferred.t -> 'a t
end

module Await_eff = Reflection(Await)
open Term(Await)

let handle_await f =
  let rec loop = function
    | Return x -> Deferred.return x
    | Op(Await d, k) ->
        Deferred.bind d (fun x -> loop (k x))
  in
  loop (Await_eff.reify f)

let await h d =
  Await_eff.perform (Await d) h
```

Using this we can write direct-style code using asynchronous I/O:

```
let copy_file h src dst =
  let s = await h (File.read src) in
  await h (File.write dst s)
```

2.7 One-shot continuations

For various practical reasons, OCaml's algebraic effects are restricted to only allow continuations to be resumed a single time. This rules out interesting effects like backtracking search, but improves the ability to reason about the interaction of effects and resources.

There is no fundamental reason that effect reflection should be restricted to only one-shot continuations, so for the purposes of this paper we shall ignore the restriction. However, for the

same reasons as OCaml, our actual implementation is restricted one-shot continuations. Note that OxCaml's `once` mode [11] allows us to enforce this restriction statically.

2.8 Nested effect handlers

A key feature of algebraic effects is how easy they are to compose. We can use state whilst using asynchronous I/O by simply nesting their handlers:

```
handle_await (fun ah ->
  handle_state i (fun sh ->
    List.iter (fun fl ->
      let s = await ah (File.read fl) in
      let i = int_of_string s in
      set sh (i + get sh)) files;
    get sh))
```

This relies on the fact that the `handle_state` uses `reify_local` and so is able to operate on a local computation that closes over the local await handler value `ah`.

If we tried to do something similar using our `handle_gen` handler, then we get an error:

```
handle_await (fun ah ->
  handle_gen (fun gh ->
    List.iter (fun fl ->
      let s = await ah (File.read fl) in
      let i = int_of_string s in
      gen gh i) files))
Error: let s = await ah (File.read fl) in
                        ^^
This value is local and would escape its region.
```

We could change the definition of `handle_gen` to use `reify_local` instead of `reify`, but then we would lose the property that the generator we produce can be passed around freely. Instead we need to build a handler that can forward the `Await` operation to different effect handlers at different points in the program's execution.

2.9 Effect sums and effect forwarding

Composing handlers, like `handle_gen`, that use `reify` rather than `reify_local` is a fundamentally more involved operation. Consider the case of generators that are able to use asynchronous I/O. We are given a computation that performs both `Await` and `Gen` operations and we'd like to turn it into a stream of integers. We would like to be able to freely pass this stream around. In particular, we do not wish the stream to be prevented from escaping the scope of the current `Await` handler. However forcing parts of the stream requires performing additional `Await` operations, so whenever we force part of the stream we'll need to provide a handler for those `Await` operations again.

To achieve this more general form of composition, we require an additional capability in our effect reflection API. We need to be able to manipulate `Handler` types for sums of effects.

```
module Sum (L : Op) (R : Op) : sig
  type 'a t =
    | Left : 'a L.t -> 'a t
    | Right : 'a R.t -> 'a t
end
```

```
module Project(L : Op) (R : Op) : sig
  val outl : Handler(Sum(L)(R)).t @ local -> Handler(L).t @ local

  val outr : Handler(Sum(L)(R)).t @ local -> Handler(R).t @ local
end
```

`Sum(L)(R)` is the coproduct of the effects `L` and `R`. Terms in the free algebras of the sum of two effects (i.e. `Term(Sum(L)(R)).t`) are isomorphic to a recursive datatype that is built from the terms of one of the effects and the operations of the other [8]. We can define this type and associated isomorphism as:

```
module Half_term (L : Op) (R : Op) : sig
  type 'a t =
    | Return : 'a @@ global -> 'a t
    | Op : 'r R.t @@ global * ('r -> Handler(L).t @ local -> 'a t) -> 'a t
end

module Half_reflection (L : Op) (R : Op) : sig
  val reify :
    (Handler(L).t @ local -> Handler(R).t @ local -> 'a)
    -> Handler(L).t @ local -> 'a Half_term(L)(R).t

  val reflect :
    (Handler(L).t @ local -> 'a Half_term(L)(R).t)
    -> Handler(L).t @ local -> Handler(R).t @ local -> 'a
end
```

Note that the key difference between `Half_term(L)(R).t` and `Term(R).t` is that continuing the computation after an operation requires passing in a local `Handler(L).t` again.

Now we can use this machinery to conveniently build a handler of `Gen` that also forwards operations from `Await`:

```
module Await_gen_eff = Half_reflection(Await)(Gen)
open Half_term(Await)(Gen)

type aints =
  | Finished
  | More of int * (Handler(Await).t @ local -> aints)

let handle_gen_in_await f ah =
    let rec loop = function
    | Return () -> Finished
    | Op(Gen i, k) -> More(i, fun ah -> loop (k () ah))
  in
  loop (Await_gen_eff.reify f ah)
```

where `handle_gen_in_await` has the type:

```
val handle_gen_in_await :
  (Handler(Await).t @ local -> Handler(Gen).t @ local -> unit)
  -> Handler(Await).t @ local
```

```
  -> aints
```

The literature on lexical effect handlers often ignores this issue. For example, Xie et al. [16] proposed a system which only supported directly nesting effect handlers and had no mechanism for writing a composable handler for generators that could be moved between different underlying asynchronous I/O handlers.

This is because composition requiring forwarding is much less common than nesting and it fundamentally requires each use of the `Await` operation to perform work proportional to the number of handlers it is forwarded through – often referred to as a "linear search". There has been a tendency to treat this linear search as some form of deficiency of algebraic effect handlers, but it is simply a classic trade-off between the extra expressivity allowed by an indirection and the runtime cost of traversing that indirection. With our effect reflection API we are able to use nesting for the cases that it supports – avoiding the cost of the indirection – and fall back to forwarding for the more general cases.

2.10 Parameterized effects and handlers

The state handlers in this paper so far have all been restricted to `int` state. This is a limitation of the effect reflection API that we have been using. We would like to support *parameterized effects* – allowing a single handler to be used with a whole family of effects. For example, a state handler parameterized by the type of the state.

We can support parameterized effects by adding a module type for them, along with parameterized versions of `Term` and `Handler`:

```
module type Op1 = sig type ('a, 'e) t end

module Term1 (O : Op1) : sig
  type ('a, 'e) t =
    | Return : 'a @@ global -> ('a, 'e) t
    | Op : ('r, 'e) O.t @@ global * ('r -> ('a, 'e) t) -> ('a, 'e) t
end

module Handler1(O : Op1) : sig
  type 'e t
end
```

with a corresponding parameterized version of `Reflection`. We can then define a parameterized state effect:

```
module State = struct
  type ('a, 's) t =
    | Get : ('s, 's) t
    | Set : 's -> (unit, 's) t
end
```

and write a handler that provides a parameterized interface for the state effect:

```
type 's handler

val handle_state : 's -> ('s handler @ local -> 'a) @ local -> 'a
val get : 's handler @ local -> 's
val set : 's handler @ local -> 's -> unit
```

$$\begin{aligned}
\text{types, } \tau &::= \tau \times \tau \mid 1 \mid \tau + \tau \mid \tau \to \tau \mid \dots \\
\text{contexts, } \Gamma &::= \varepsilon \mid \Gamma;\, x : \tau \mid \dots \\
\text{values, } v &::= x \mid (v, v) \mid \pi_1\, v \mid \pi_2\, v \mid () \mid \mathrm{in}_L\, v \mid \mathrm{in}_R\, v \mid \lambda(x : \tau).\, c \mid \dots \\
\text{computations, } c &::= \mathrm{return}\, v \mid \mathrm{let}\, x = c\, \mathrm{in}\, c \mid v\, v \mid \\
&\qquad \mathrm{case}\, v\, \{\, \mathrm{in}_L\, x \Rightarrow c\, ;\, \mathrm{in}_R\, x \Rightarrow c\, \} \mid \dots
\end{aligned}$$

$$\frac{}{\Gamma;\, x : \tau \vdash_V x : \tau} \qquad \frac{\Gamma \vdash_V v_1 : \tau_1 \qquad \Gamma \vdash_V v_1 : \tau_1}{\Gamma \vdash_V (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash_V v : \tau_1 \times \tau_2}{\Gamma \vdash_V \pi_1\, v : \tau_1} \qquad \frac{\Gamma \vdash_V v : \tau_1 \times \tau_2}{\Gamma \vdash_V \pi_2\, v : \tau_2}$$

$$\frac{}{\Gamma \vdash_V () : 1} \qquad \frac{\Gamma \vdash_V v : \tau_1}{\Gamma \vdash_V \mathrm{in}_L\, v : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash_V v : \tau_2}{\Gamma \vdash_V \mathrm{in}_R\, v : \tau_1 + \tau_2} \qquad \frac{\Gamma;\, x : \tau_1 \vdash_C c : \tau_2}{\Gamma \vdash_V \lambda(x : \tau_1).\, c : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_V v : \tau}{\Gamma \vdash_C \mathrm{return}\, v : \tau} \qquad \frac{\Gamma \vdash_C c_1 : \tau_1 \qquad \Gamma;\, x : \tau_1 \vdash_C c_2 : \tau_2}{\Gamma \vdash_C \mathrm{let}\, x = c_1\, \mathrm{in}\, c_2 : \tau_2} \qquad \frac{\Gamma \vdash_V v_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_V v_2 : \tau_1}{\Gamma \vdash_C v_1 v_2 : \tau_2}$$

$$\frac{\Gamma \vdash_V v : \tau_1 + \tau_2 \qquad \Gamma;\, x_1 : \tau_1 \vdash_C c_1 : \tau_3 \qquad \Gamma;\, x_2 : \tau_2 \vdash_C c_2 : \tau_3}{\Gamma \vdash_C \mathrm{case}\, v\, \{\, \mathrm{in}_L\, x_1 \Rightarrow c_1\, ;\, \mathrm{in}_R\, x_2 \Rightarrow c_2\, \} : \tau_3}$$

Fig. 1. Basic syntax and typing rules

The need for a new copy of the interface for each type constructor arity is an artefact of OCaml's approach to higher-kinded types. Languages with direct support for higher-kinded types should be able to support parameterized effects of arbitrary arity with a single interface.

3 A language with effect reflection

The design of effect reflection falls out naturally from a denotational semantics in terms of presheaves. To make this connection precise and demonstrate the key properties of effect reflection, we present a simply-typed calculus that directly incorporates the constructs needed for effect reflection.

3.1 The base language

Our language is simply typed, featuring all the usual type-formers. It uses a variation of fine-grained call-by-value: it has separate syntax and typing judgements for values and computations. The syntax and typing rules for the base language are shown in Fig.1.

3.2 Locality

We track locality in the language via a global comonadic modality $\Box$. In our setting being global means not depending on the ambient monad. For example, $1 \to 1$ represents computations that may use operations from the ambient monad, while $\Box(1 \to 1)$ contains only pure computations that ignore the ambient monad.

When using box v to build a value with a global type, we can only use existing values that also have a global type. We achieve this using a fairly standard approach for modal types: we extend contexts with a binding form that requires a global type and then we restrict the context in which we type v to only include such bindings. The new global binding form is written $\Gamma;\, x :_{\Box} \tau$ and

$$\begin{aligned}
\text{types, } \tau &::= \ldots \mid \Box\,\tau \mid \ldots \\
\text{contexts, } \Gamma &::= \ldots \mid \Gamma;\, x :_{\Box} \tau \\
\text{values, } v &::= \ldots \mid \mathrm{box}\, v \mid \mathrm{let}(\mathrm{box}\, x) = v \,\mathrm{in}\, v \mid \mathrm{run}(c) \mid \ldots
\end{aligned}$$

$$\frac{}{\Gamma;\, x :_{\Box} \tau \vdash_V x : \tau} \qquad \frac{\Gamma/\Box \vdash_V v : \tau}{\Gamma \vdash_V \mathrm{box}\, v : \Box\,\tau} \qquad \frac{\Gamma \vdash_V v_1 : \Box\,\tau_1 \qquad \Gamma;\, x :_{\Box} \tau_1 \vdash_V v_2 : \tau_2}{\Gamma \vdash_V \mathrm{let}(\mathrm{box}\, x) = v_1 \,\mathrm{in}\, v_2 : \tau_2}$$

$$\frac{\Gamma/\Box \vdash_C c : \tau}{\Gamma \vdash_V \mathrm{run}(c) : \tau}$$

Fig. 2. Locality syntax and typing rules

represents a value in the context of type $\Box\,\tau$. We create global bindings in the context using the elimination form: $\mathrm{let}(\mathrm{box}\, x) = v_1 \,\mathrm{in}\, v_2$.

Restricting the context is implemented via an operation on contexts $\Gamma/\Box$ that removes all local bindings, leaving only the global ones:

$$\begin{aligned}
\varepsilon/\Box &= \varepsilon \\
(\Gamma;\, x : \tau)/\Box &= \Gamma/\Box \\
(\Gamma;\, x :_{\Box} \tau)/\Box &= \Gamma/\Box;\, x :_{\Box} \tau
\end{aligned}$$

Global computations do not depend on the ambient monad, so we also include a value form $\mathrm{run}(c)$, which allows treating a global computation c as a value.

The syntax and typing rules related to locality are in Fig.2

3.3 Signatures and inductive types

Effect reflection requires, for any signature Σ that is being reflected, an inductive type W_Σ to represent the free algebras of Σ. W_Σ is sometimes called the *freer monad* of Σ [9].

As is standard for algebraic effects systems, we only support handlers for algebraic theories without any equations, even though our semantics is defined in terms of arbitrary theories. We define algebraic signatures Σ as finite sets of operations:

$$\{op_1 : \tau_{p_1} \rightarrowtail \tau_{a_1}; \ldots; op_n : \tau_{p_n} \rightarrowtail \tau_{a_n}\}$$

An operation type $\tau_p \rightarrowtail \tau_a$ indicates an operation parameterized by τ_p and with arity τ_a.

Example 3.1. The signature for boolean state, which we'll call Σ_{bs} is:

$$\{\mathrm{get} : 1 \rightarrowtail (1+1); \mathrm{set} : (1+1) \rightarrowtail 1\}$$

We also allow the sum of two signatures $\Sigma_1 + \Sigma_2$ to be used as a signature. It has operations:

$$\begin{aligned}
\{\ &\mathrm{left}(op_{1,1}) : \tau_{p_{1,1}} \rightarrowtail \tau_{a_{1,1}}\ ;\ \ldots\ ;\ \mathrm{left}(op_{1,n}) : \tau_{p_{1,n}} \rightarrowtail \tau_{a_{1,n}}\ ; \\
&\mathrm{right}(op_{2,1}) : \tau_{p_{2,1}} \rightarrowtail \tau_{a_{2,1}}\ ;\ \ldots\ ;\ \mathrm{right}(op_{2,m}) : \tau_{p_{2,m}} \rightarrowtail \tau_{a_{2,m}}\ \}
\end{aligned}$$

where $op_{1,1}\ ;\ \ldots\ ;\ op_{1,n}$ are the operations of Σ_1 and $op_{2,1}\ ;\ \ldots\ ;\ op_{2,m}$ are the operations of Σ_2.

For each operation $op : \tau_p \rightarrowtail \tau_a$, $W_\Sigma(\tau)$ has a constructor $op_\Sigma(v_p, v_k)$ where $v_p : \Box\,\tau_p$ and $v_k : \Box\,\tau_a \to W_\Sigma(\tau)$. $W_\Sigma(\tau)$ also has a constructor $\mathrm{ret}_\Sigma(v)$ where $v : \tau$.

$$\begin{aligned}
\text{types, } \tau &::= \ldots \mid W_\Sigma(\tau) \mid \ldots \\
\text{signatures, } \Sigma &::= \{op : \tau \rightarrowtail \tau\,;\, \ldots\,;\, op : \tau \rightarrowtail \tau\} \mid \Sigma + \Sigma \\
\text{values, } v &::= \ldots \mid op_\Sigma(v,\, v) \mid \mathrm{ret}_\Sigma(v) \mid \ldots \\
\text{computations, } c &::= \ldots \mid \mathrm{rec}_\Sigma\, v\, \{\mathrm{ret}(x) \Rightarrow c\,;\, op(x,x) \Rightarrow c\,;\, \ldots\,;\, op(x,x) \Rightarrow c\} \mid \ldots
\end{aligned}$$

$$\frac{op : \tau_1 \rightarrowtail \tau_2 \in \Sigma \qquad \Gamma \vdash_V v_1 : \Box\, \tau_1 \qquad \Gamma \vdash_V v_2 : \Box\, \tau_2 \to W_\Sigma(\tau)}{\Gamma \vdash_V op_\Sigma(v_1,\, v_2) : W_\Sigma(\tau)} \qquad \frac{\Gamma \vdash_V v : \tau}{\Gamma \vdash_V \mathrm{ret}_\Sigma(v) : W_\Sigma(\tau)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_V v : W_\Sigma(\tau_1) \qquad \Sigma = \{op_1 : \tau_{p_1} \rightarrowtail \tau_{a_1}; \ldots; op_n : \tau_{p_n} \rightarrowtail \tau_{a_n}\} \qquad \Gamma;\, x_r : \tau_1 \vdash_C c_r : \tau_2 \\ \Gamma;\, x_{p_1} : \Box\, \tau_{p_1};\, x_{k_1} : \Box\, \tau_{a_1} \to \tau_2 \vdash_C c_1 : \tau_2 \qquad \ldots \qquad \Gamma;\, x_{p_n} : \Box\, \tau_{p_n};\, x_{k_n} : \Box\, \tau_{a_n} \to \tau_2 \vdash_C c_n : \tau_2\end{array}}{\Gamma \vdash_C \mathrm{rec}_\Sigma\, v\, \{\mathrm{ret}(x_r) \Rightarrow c_r\,;\, op_1(x_{p_1}, x_{k_1}) \Rightarrow c_1\,;\, \ldots\,;\, op_n(x_{p_n}, x_{k_n}) \Rightarrow c_n\} : \tau_2}$$

Fig. 3. Inductive type syntax and typing rules

The eliminator for W_Σ is written

$$\begin{aligned}
&\mathrm{rec}_\Sigma\, v\, \{\mathrm{ret}(x) \Rightarrow c_r\,; \\
&\qquad op_1(x_{p_1}, x_{k_1}) \Rightarrow c_1\,; \\
&\qquad \ldots\,; \\
&\qquad op_n(x_{p_n}, x_{k_n}) \Rightarrow c_n\}
\end{aligned}$$

The syntax and typing rules for inductive types are shown in Fig.3.

Example 3.2. The following value of type $W_{\Sigma_{bs}}$ represents a reified computation that negates the value of the state:

$$\begin{aligned}
&get_{\Sigma_{bs}}((), \\
&\quad \lambda(x_a : 1 + 1). \\
&\qquad \mathrm{let}\, x_n = negate\, x_a\, \mathrm{in} \\
&\qquad set_{\Sigma_{bs}}(x_n,\, \lambda(x_u : 1).\, \mathrm{ret}_{\Sigma_{bs}}(x_u)))
\end{aligned}$$

where *negate* is the function that negates a boolean.

3.4 Effect reflection

The language provides a type $y(\Sigma)$ for each Σ to represent mappings from the operations of Σ into the current ambient monad. Effect reflection appears in the language as two constructs: reflection and reification. Reflection $\Uparrow (v(op))$ uses a value of type $y(\Sigma)$ to map the operation op from Σ into the ambient monad. Reification $\Downarrow_\Sigma$ extends the ambient monad with the operations from Σ, providing a value of type $y(\Sigma)$ for constructing those operations via reflection. These constructs implement the core isomorphism of effect reflection: reification can convert functions $y(\Sigma) \to \Box\, \tau$ into terms $W_\Sigma(\Box\, \tau)$, while reflection provides the inverse direction. Their syntax and typing rules are shown in Fig.4.

$$\begin{aligned}
\text{types, } \tau &::= \ldots \mid y(\Sigma) \mid \ldots \\
\text{values, } v &:= \ldots \mid \mathrm{out}_L\, v \mid \mathrm{out}_R\, v \\
\text{computations, } c &:= \ldots \Uparrow (v(op))(v, x.c) \mid \Downarrow_\Sigma (x.\, c)
\end{aligned}$$

$$\frac{\Gamma \vdash_V v : y(\Sigma_1 + \Sigma_2)}{\Gamma \vdash_V \mathrm{out}_L\, v : y(\Sigma_1)} \qquad \frac{\Gamma \vdash_V v : y(\Sigma_1 + \Sigma_2)}{\Gamma \vdash_V \mathrm{out}_R\, v : y(\Sigma_2)}$$

$$\frac{\Gamma \vdash_V v_1 : y(\Sigma) \qquad op : \tau_1 \rightarrowtail \tau_2 \in \Sigma \qquad \Gamma \vdash_V v_2 : \Box\, \tau_1 \qquad \Gamma;\, x : \Box\, \tau_2 \vdash_C c : \tau_3}{\Gamma \vdash_C \Uparrow (v_1(op))(v_2, x.c) : \tau_3}$$

$$\frac{\Gamma;\, x : y(\Sigma) \vdash_C c : \Box\, \tau}{\Gamma \vdash_C \Downarrow_\Sigma (x.\, c) : W_\Sigma(\Box\, \tau)}$$

Fig. 4. Effect reflection syntax and typing rules

Note that we treat reflection of individual operations $\Uparrow (v(op))$ as primitive. We can define reflection of terms $\Uparrow_\Sigma$ in terms of $\Uparrow (v(op))$:

$$\begin{aligned}
\Uparrow_\Sigma(v, c) := \quad & \mathrm{let}\, x_w = c\, \mathrm{in} \\
& \mathrm{rec}_\Sigma\, x_w \,\{ \\
& \quad \mathrm{ret}(x_r) \Rightarrow \mathrm{return}\, x_r\, ; \\
& \quad op_1(x_{p_1}, x_{k_1}) \Rightarrow \Uparrow (v(op_1))(x_{p_1}, x_{a_1}.(x_{k_1}\, x_{a_1}))\, ; \\
& \quad \ldots\, ; \\
& \quad op_n(x_{p_n}, x_{k_n}) \Rightarrow \Uparrow (v(op_n))(x_{p_n}, x_{a_n}.(x_{k_n}\, x_{a_n}))\}
\end{aligned}$$

which has this admissible typing rule:

$$\frac{\Gamma \vdash_V v : y(\Sigma) \qquad \Gamma \vdash_V c : W_\Sigma(\Box\, \tau)}{\Gamma \vdash_C \Uparrow_\Sigma(v, c) : \Box\, \tau}$$

We also provide projections out_L and out_R that decompose a mapping for $\Sigma_1 + \Sigma_2$ into mappings for Σ_1 and Σ_2 respectively.

Example 3.3. The value from Example 3.2 can be constructed via reflection as:

$$\begin{aligned}
& \Downarrow_{\Sigma_{bs}} (x_h. \\
& \quad \Uparrow (x_h(get))((), x_a. \\
& \qquad \mathrm{let}\, x_n = \mathit{negate}\, x_a\, \mathrm{in} \\
& \qquad \Uparrow (x_h(set))(x_n, x_u.\, \mathrm{return}\, x_u)))
\end{aligned}$$

where *negate* is the function that negates a boolean.

3.5 Operational semantics

We define the operational semantics as a pair of mutually defined reduction relations $\rightsquigarrow_V$ and $\rightsquigarrow_C$ for values and computations respectively shown in Fig.5. The definitions of normal forms for values and computations are given in Fig.6. Subject reduction and strong normalization of these rules can be proved using standard techniques.

$$
\begin{aligned}
\pi_1\,(v_1, v_2) &\rightsquigarrow_V v_1 \\
\pi_2\,(v_1, v_2) &\rightsquigarrow_V v_2 \\
\mathrm{let}(\mathrm{box}\,x) = \mathrm{box}\,v_1\,\mathrm{in}\,v_2 &\rightsquigarrow_V v_2[x\backslash v_1] \\
\mathrm{run}(\mathrm{return}\,v) &\rightsquigarrow_V v \\
\mathrm{let}\,x = \mathrm{return}\,v\,\mathrm{in}\,c &\rightsquigarrow_V c[x\backslash v] \\
\mathrm{let}\,x_1 = \Uparrow\,(v_1(op))(v_2, x_2.c_1)\,\mathrm{in}\,c_2 &\rightsquigarrow_V \Uparrow\,(v_1(op))(v_2, x_2.\,\mathrm{let}\,x_1 = c_1\,\mathrm{in}\,c_2) \\
(\lambda(x : \tau).\,c)\,v &\rightsquigarrow_C c[x\backslash v] \\
\mathrm{case}(\mathrm{in}_L\,v)\,\{\,\mathrm{in}_L\,x_1 \Rightarrow c_1\,;\,\mathrm{in}_R\,x_2 \Rightarrow c_2\,\} &\rightsquigarrow_C c_1[x_1\backslash v] \\
\mathrm{case}(\mathrm{in}_R\,v)\,\{\,\mathrm{in}_L\,x_1 \Rightarrow c_1\,;\,\mathrm{in}_R\,x_2 \Rightarrow c_2\,\} &\rightsquigarrow_C c_2[x_2\backslash v] \\
\mathrm{rec}_\Sigma(\mathrm{ret}_\Sigma(v))\,\{\mathrm{ret}(x) \Rightarrow c\,;\,\ldots\} &\rightsquigarrow_C c[x\backslash v] \\
\mathrm{rec}_\Sigma\,op_\Sigma(v_1,\,v_2)\,\{H_1\,;\,op(x_1, x_2) \Rightarrow c\,;\,H_2\} & \\
\rightsquigarrow_C c[x_1\backslash v_1][x_2\backslash\,\lambda(x_a : \tau_a).\,\mathrm{let}\,x_w = v_2\,x_a\,\mathrm{in}\,\mathrm{rec}_\Sigma\,x_w\,\{H_1\,;\,op(x_1, x_2) \Rightarrow c\,;\,H_2\}] & \\
\Downarrow_\Sigma(x.\ \mathrm{return}\,v) &\rightsquigarrow_C \mathrm{return}\,\mathrm{ret}_\Sigma(v) \\
\Downarrow_\Sigma(x_1.\ \Uparrow\,(x_1(op))(v, x_2.c)) &\rightsquigarrow_C \mathrm{return}\,op_\Sigma(v,\,\lambda(x_2 : \Downarrow_\Sigma(x_1.\,c)).\,) \\
\Downarrow_\Sigma(x_1.\ \Uparrow\,(x_2(op))(v, x_3.c)) &\rightsquigarrow_C \Uparrow\,(x_2(op))(v, x_3.\Downarrow_\Sigma(x_1.\,c)) \qquad (x_1 \neq x_2) \\
\Uparrow\,((\mathrm{out}_L\,v_1)(op))(v_2, x.c) &\rightsquigarrow_C \Uparrow\,(v_1(\mathrm{left}(op)))(v_2, x.c) \\
\Uparrow\,((\mathrm{out}_R\,v_1)(op))(v_2, x.c) &\rightsquigarrow_C \Uparrow\,(v_1(\mathrm{right}(op)))(v_2, x.c)
\end{aligned}
$$

Fig. 5. Reduction relations

$$
\begin{aligned}
\text{neutral values, } \eta &:= x \mid \pi_1\,\eta \mid \pi_2\,\eta \mid \mathrm{let}(\mathrm{box}\,x) = \eta\,\mathrm{in}\,\omega \mid \mathrm{run}(\mu) \\
\text{neutral computations, } \mu &:= \mathrm{let}\,x = \mu\,\mathrm{in}\,\gamma \mid \eta\,\omega \mid \\
&\quad\ \mathrm{case}\,\eta\,\{\,\mathrm{in}_L\,x \Rightarrow \gamma\,;\,\mathrm{in}_R\,x \Rightarrow \gamma\,\} \mid \\
&\quad\ \mathrm{rec}_\Sigma\,\eta\,\{\mathrm{ret}(x) \Rightarrow \gamma\,;\,op(x, x) \Rightarrow \gamma\,;\,\ldots\,;\,op(x, x) \Rightarrow \gamma\} \\
&\quad\ \Downarrow_\Sigma(x.\,\mu) \\
\text{normal values, } \omega &:= \eta \mid (\omega, \omega) \mid () \mid \mathrm{in}_L\,\omega \mid \mathrm{in}_R\,\omega \mid \lambda(x : \tau).\,\gamma \mid \\
&\quad\ op_\Sigma(\omega,\,\omega) \mid \mathrm{ret}_\Sigma(\omega) \mid \\
&\quad\ \mathrm{box}\,\omega \mid \mathrm{out}_L\,\omega \mid \mathrm{out}_R\,\omega \\
\text{normal computations, } \gamma &:= \mu \mid \mathrm{return}\,\omega \mid \Uparrow\,(\eta(op))(\omega, x.\gamma)
\end{aligned}
$$

Fig. 6. Normal forms

Note that the reduction relations do not include η expansion. They also do not include some conversions you might expect for computations:

- There is no rule to convert $\mathrm{let}\,x_2 = \mathrm{let}\,x_1 = c_1\,\mathrm{in}\,c_2\,\mathrm{in}\,c_3$ to $\mathrm{let}\,x_1 = c_1\,\mathrm{in}\,\mathrm{let}\,x_2 = c_2\,\mathrm{in}\,c_3$ or vice versa.
- There is no rule to convert $\mathrm{let}\,x = c\,\mathrm{in}\,\mathrm{return}\,x$ to c.

These rules are excluded because they are not β rules. Instead they are better thought of as ν rules [1]: additional equalities between neutral terms that are immediate consequences of structural

$$
\begin{array}{ll}
\wr 1 \int = 1 & \wr x \int = x \\
\wr A \times B \int = \wr A \int \times \wr B \int & \wr (e, f) \int = (\wr e \int, \wr f \int) \\
\wr A \to B \int = (\Box \wr A \int) \to \wr B \int & \wr \pi_1\, e \int = \pi_1\, \wr e \int \\
 & \wr \pi_2\, e \int = \pi_2\, \wr e \int \\
\wr \varepsilon \int = \varepsilon & \wr \lambda (x : A).\, e \int = \lambda (x' : \Box \wr A \int).\ \mathrm{return}(\mathrm{let}(\mathrm{box}\, x) = x' \,\mathrm{in} \wr e \int) \\
\wr \Gamma;\, x : A \int = \wr \Gamma \int;\, x :_{\Box} \wr A \int & \wr f\, e \int = \mathrm{run}(\wr f \int\, (\mathrm{box} \wr e \int))
\end{array}
$$

Fig. 7. Embedding of Simply-Typed Lambda Calculus into values

induction. Rather than be included as part of ordinary reduction they can be done as an additional standardization pass on neutral terms after reduction and η-expansion have completed. This staged approach to computing equality of terms works because ν rules cannot produce additional β redexes.

3.6 Embedding the Simply-Typed Lambda Calculus

Using the global modality and run(c) construct we can embed the Simply Typed Lambda Calculus (STLC) into this language as values. This demonstrates that the language is a proper extension of STLC. We define an embedding $\wr \_ \int$ on types, contexts and terms of STLC in Fig.7. Note that $\rightsquigarrow_V$ matches the behaviour of β reduction of STLC too, with each STLC reduction step corresponding to up to 3 steps of $\rightsquigarrow_V$.

4 Presheaves for algebraic effects

The design of effect reflection falls out naturally from a simple denotational semantics built on presheaves. This semantic foundation not only validates our approach but also makes the connection between locality and effects precise. The key idea is to index the meaning of types by an ambient algebraic monad of effects: rather than have the semantics of each type be a set of elements, the semantics of each type is a mapping from the ambient monad to a set of elements.

4.1 Algebraic monads

In order to index the meaning of our types by the ambient monad we need some way of representing that ambient monad. As with all approaches to algebraic effects, we must restrict ourselves to monads that have some nice properties, in particular we need the sum of two of these monads to always exists.

To this end, we will restrict the ambient monad to be an *algebraic monad*: one that can be expressed as the free algebras of an algebraic theory. Note that this definition of algebraic monad is not precise: different choices for the definition of "algebraic theory" give you different meanings for "algebraic monad". We'll make our particular choice over the course of this section.

We recall the definitions of algebraic signature, terms of a signature, algebraic theory and models of a theory.

Definition 4.1 (Algebraic signature)**.** An *algebraic signature* is a set of operations with associated arities.

Example 4.1. The algebraic signature for boolean state Σ_{bs} is:

$$\{(\mathrm{get}, 2); (\mathrm{set\_true}, 1); (\mathrm{set\_false}, 1)\}$$

Definition 4.2 (Term)**.** A *term* t of some algebraic signature Σ is either a free variable x or $op(t_1, \ldots, t_k)$ where op is an operation of Σ with arity k and each t_i is itself a term.

Definition 4.3 (Algebraic theory)**.** An *algebraic theory* consists of an algebraic signature Σ and a set of equations on terms of Σ.

Example 4.2. The algebraic theory for boolean state L_{bs} adds the following equations to the signature of boolean state Σ_{bs}:

$$\begin{aligned}
&\text{set_true}(\text{get}(x, y)) = \text{set_true}(x) && \text{set_false}(\text{get}(x, y)) = \text{set_false}(y)\\
&\text{set_true}(\text{set_true}(x)) = \text{set_true}(x) && \text{set_true}(\text{set_false}(x)) = \text{set_false}(x)\\
&\text{set_false}(\text{set_true}(x)) = \text{set_true}(x) && \text{set_false}(\text{set_false}(x)) = \text{set_false}(x)\\
&\text{get}(\text{set_true}(x), \text{set_false}(x))) = x
\end{aligned}$$

Definition 4.4 (Model)**.** A *model* M of some algebraic theory L consists of a carrier set S and for each operation op in the signature of L a function $op_M : (\Pi_k S) \to S$ where k is the arity of op, such that the equations of L are respected when interpreting the terms using these functions for the operations. Models of L are also called algebras of L.

Example 4.3. We can model the theory of boolean state L_{bs} using the carrier set $2 \to 2$ and interpreting the operations with:

$$\begin{aligned}
\text{get}(x, y) &= \lambda s.\ \text{if } s \text{ then } x(s) \text{ else } y(s)\\
\text{set_true}(x) &= \lambda s.\, x(\text{true})\\
\text{set_false}(x) &= \lambda s.\, x(\text{false})
\end{aligned}$$

Arities in algebraic signatures are often considered as natural numbers, but they can equivalently be just sets, with families of terms indexed by that set used as the arguments of that operator in a term. It is also convenient to allow families of operations parameterized by some set. As such, we can present a signature in the form:

$$\{op_1 : P_1 \rightarrowtail A_1; \ldots; op_n : P_n \rightarrowtail A_n\}$$

where $op : P \rightarrowtail A$ indicates an operation op parameterized by P and with arity A.

Example 4.4. The algebraic signature for boolean state Σ_{bs} can be written as:

$$\{\text{get} : 1 \rightarrowtail 2; \text{set} : 2 \rightarrowtail 1\}$$

Given an algebraic theory L and a set A we can construct the free model, or *free algebra*, $T_L(A)$. The values $T_L(A)$ are terms built using the elements of A and the operations of L quotiented by the equations in L. T_L is a monad and T is one half of an equivalence between algebraic theories and algebraic monads.

Example 4.5. The free algebras of the signature of boolean state $T_{\Sigma_{bs}}$ are equivalent to the type:

```
type 'a t =
  | Return of 'a
  | Get of unit * (bool -> 'a t)
  | Set of bool * (unit -> 'a t)
```

The free algebras of the theory of boolean state $T_{L_{bs}}$ are equivalent to the above type quotiented by the equations of L_{bs}.

For an algebraic signature Σ the free algebra T_Σ is the free monad of an endofunctor F_Σ. The values of $F_\Sigma(A)$ are of the form $op(a_1, \ldots, a_k)$ where op is an operation of Σ with arity k and each a_i is an element of A.

Example 4.6. $F_{\Sigma_{bs}}$ is equivalent to the type:

```
type 'a t =
  | Get of unit * (bool -> 'a)
  | Set of bool * (unit -> 'a)
```

Traditional algebraic signatures are restricted to the case of a finite set of operations each with a finite arity, but programming languages require more generality. $\mathbb{N}$ state needs countable arities, while $\mathbb{N} \to \mathbb{N}$ state requires uncountable arities. For these we must extend algebraic signatures to allow for arbitrary sets of operations with arbitrary sets as arities. Such algebraic theories are called *bounded infinitary algebraic theories*. They are "bounded" because for each such theory there is some regular cardinal κ that is a bound on the arities of the theory. An algebraic theory with some bound κ corresponds to a monad with rank κ. Thus we take monads with a rank as our precise definition of algebraic monad.

Most monads used in practice for programming are algebraic by this definition, with the notable exception of the continuation monad. The issue with trying to represent non-algebraic monads algebraically is essentially one of size: if you try to write down the collection of algebraic operations for the continuation monad you get something that is too large to be a set. This is closely related to the strict positivity restriction on inductive type definitions.

4.2 (Bounded) Lawvere theories

We use *Lawvere theories* as our concrete representation of the ambient algebraic monad. Lawvere theories are a categorical representation of algebraic theories that doesn't depend on the particular choice of operations and equations.

Definition 4.5. A *Lawvere theory* is a category L with finite products in which every object is isomorphic to some finite cartesian product $X^n = X \times \ldots \times X$ of a distinguished object X.

The objects of a Lawvere theory correspond to the different possible arities and the morphisms $X^n \to X$ correspond to terms of L with n free variables quotiented by the equations in L. A model of a Lawvere theory L is a product-preserving functor $L \to \mathrm{Set}$.

Lawvere theories correspond to the traditional form of algebraic theories: those with a finite set of operations with finite arities, but we can extend the definition to get something that corresponds to *bounded infinitary algebraic theories*.

Definition 4.6. A *Bounded Lawvere theory* is a small category L with small products in which every object is isomorphic to some small cartesian product $\Pi_S X$ of a distinguished object X.

Note that the "bounded" aspect of the definition comes from the restriction that the category be small.

We will mostly describe Lawvere theories and constructions on them in terms of their presentations as algebraic theories, so an understanding the details of Lawvere theories is not required for understanding the rest of this paper.

Bounded Lawvere theories form a category Law whose morphisms are equivalent to maps from the operations of one theory to the operations of another that preserve the equations of the first theory according to the equations of the second.

Law has an initial object $\bot$, which is the algebraic theory with no operations. It also has coproducts $L + L'$ which is the theory made by unioning the operations and equations of L and L'.

4.3 Presheaves

For a category $\mathbb{C}$, $\widehat{\mathbb{C}} = \mathrm{Set}^{\mathbb{C}^{\mathsf{op}}}$ is the category of presheaves over $\mathbb{C}$. Its objects are contravariant functors from $\mathbb{C}$ to Set and it's morphisms are natural transformations between such functors.

A common use of presheaves is where $\mathbb{C}$ has finite products and is thought of as some form of context over which everything is parameterized. We use presheaves over Law because types should be parameterized by the ambient algebraic theory–but contravariantly, since we want to be parameterized by theories that can be handled by the ambient monad.

As such, we use

$$\widehat{\mathrm{Law}^{\mathsf{op}}} = \mathrm{Set}^{\mathrm{Law}}$$

as the basis of our semantics. The op in the definition of presheaves and the op due to our dependence being contravariant cancel out, but it is still best to think in terms of presheaves as that is a more precise description of what we are doing.

Our semantics will use objects from $\widehat{\mathrm{Law}^{\mathsf{op}}}$ to represent types and contexts, and morphisms in $\widehat{\mathrm{Law}^{\mathsf{op}}}$ to represent values. This allows their meanings to be parameterized by the ambient monad. By using presheaves rather than ordinary functions we ensure terms and types at a particular ambient monad can always be mapped naturally to a larger ambient monad.

$$[\![\tau]\!] : \widehat{\mathrm{Law}^{\mathsf{op}}} \qquad [\![\Gamma]\!] : \widehat{\mathrm{Law}^{\mathsf{op}}} \qquad [\![\Gamma \vdash_V v : \tau]\!] : \mathrm{Hom}_{\widehat{\mathrm{Law}^{\mathsf{op}}}}([\![\Gamma]\!], [\![\tau]\!])$$

Categories of presheaves have products and coproducts, which are defined pointwise. We will use these as the semantics of product types and sum types.

$$[\![1]\!] = 1 \qquad [\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!] \qquad [\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] + [\![\tau_2]\!]$$

The semantics of the corresponding introduction and elimination forms follows the usual categorical semantics of such types.

We also use products for the semantics of contexts:

$$[\![\varepsilon]\!] = 1 \qquad [\![\Gamma; x : \tau]\!] = [\![\Gamma]\!] \times [\![\tau]\!]$$

4.4 Representing computations: The free algebras monad

Our aim is to have a semantics where the meaning of types is indexed by the ambient monad – or more precisely by the algebraic theory whose free algebras are the ambient monad. In particular, we would like the interpretation of computation types to be applications of the ambient monad.

Given a Lawvere theory L, the underlying sets of its free algebras form a monad T_L on Set. In addition to the unit of the monad, we can construct elements of T_L using the operations of L. For an operation $op : P \rightarrowtail A$ of a Lawvere theory L and some set S, there is a function $op_{T_L} : (P \times (A \to T_L(S))) \to T_L(S)$ that constructs terms in the free algebra using op.

$T\_$ is functorial and we can use it to construct a monad $\mathcal{T}$ on $\widehat{\mathrm{Law}^{\mathsf{op}}}$ such that:

$$\mathcal{T}(P)(L) = T_L(P(L))$$

We call $\mathcal{T}$ the *free algebras monad*, and we will use it as the computation monad in our denotational semantics.

$$[\![\Gamma \vdash_C v : \tau]\!] : \mathrm{Hom}_{\widehat{\mathrm{Law}^{\mathsf{op}}}}([\![\Gamma]\!], \mathcal{T}([\![\tau]\!]))$$

$$[\![\Gamma \vdash_C \mathrm{return}\, v : \tau]\!] = \eta_{[\![\tau]\!]} \circ [\![\Gamma \vdash_V v : \tau]\!]$$

$$[\![\Gamma \vdash_C \mathrm{let}\, x = c_1 \,\mathrm{in}\, c_2 : \tau_2]\!] = \mu_{[\![\tau_2]\!]} \circ \mathcal{T}([\![\Gamma; x : \tau_1 \vdash_C c_2 : \tau_2]\!]) \circ t_{[\![\Gamma]\!],[\![\tau_1]\!]} \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash_C c_1 : \tau_1]\!] \rangle$$

where η, μ and t are the unit, multiplication and strength respectively of $\mathcal{T}$.

Categories of presheaves also have exponentials, which combine with the free algebras monad to give us our semantics for function types.

$$[\![\tau_1 \to \tau_2]\!] = \mathcal{T}([\![\tau_2]\!])^{[\![\tau_1]\!]}$$

4.5 The global modality

For any category $\mathbb{C}$, if $\mathbb{C}$ has a terminal object $\top$, we can define the global modality $\Box : [\widehat{\mathbb{C}}, \widehat{\mathbb{C}}]$ such that: $\Box(P)(L) = P(\top)$. In the case of $\widehat{\mathrm{Law}^{\mathrm{op}}}$ this becomes:

$$\Box(P)(L) = P(\bot)$$

$\Box$ forms a comonad on $\widehat{\mathrm{Law}^{\mathrm{op}}}$. $\Box$ also distributes over products: $\Box(P \times Q) = (\Box P) \times (\Box Q)$

Within our semantics, this amounts to restricting a type to only the elements which do not depend on the ambient monad. This is very similar to the purity comonad of Choudhury and Krishnaswami [5]. In particular,

$$\Box\mathcal{T}(P) \quad = \quad T_\bot \circ \Box P \quad \cong \quad \Box P$$

as elements of the free algebras of the initial algebraic theory are just elements of their carrier sets.

We use the $\Box$ comonad as our semantics for $\Box$ types and $\Box$ bindings.

$$[\![\Box\,\tau]\!] = \Box[\![\tau]\!]$$
$$[\![\Gamma;\, x :_\Box \tau]\!] = [\![\Gamma]\!] \times \Box[\![\tau]\!]$$
$$[\![\Gamma;\, x :_\Box \tau \vdash_V x : \tau]\!] = \epsilon \circ \pi_2$$
$$[\![\Gamma \vdash_V \mathrm{box}\, v : \Box\,\tau]\!] = \Box[\![\Gamma/\Box \vdash_V v : \tau]\!] \circ \eta \circ !_{\Gamma/\Box}$$
$$[\![\Gamma \vdash_V \mathrm{let}(\mathrm{box}\, x) = v_1 \,\mathrm{in}\, v_2 : \tau_2]\!] = [\![\Gamma;\, x :_\Box \tau_1 \vdash_V v_2 : \tau_2]\!] \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash_V v_1 : \Box\,\tau_1]\!]\rangle$$
$$[\![\Gamma \vdash_V \mathrm{run}(c) : \Box\,\tau]\!] = \rho_{[\![\tau]\!]} \circ \Box[\![\Gamma/\Box \vdash_C c : \tau]\!] \circ \eta \circ !_{\Gamma/\Box}$$

where ϵ and η are the counit and comultiplication of $\Box$, $!_{\Gamma/\Box} : \Gamma \to \Gamma/\Box$ is the morphism that projects the global bindings from Γ, and ρ_P is the isomorphism from $\Box\mathcal{T}(P)$ to $\Box P$.

4.6 Reflecting operations: The Yoneda embedding

We want to be able to reflect operations into the ambient monad. We can use the op_{T_L} functions to build morphisms into $\mathcal{T}$, but we require some way to represent operations of the ambient monad. We can do that by using the *Yoneda embedding*. The Yoneda embedding $y(\_)$ is a full and faithful embedding from a category $\mathbb{C}$ into its category of presheaves $\widehat{\mathbb{C}}$. When treating $\mathbb{C}$ as a form of context, the elements of $y(C)$ can be used to represent variables of type C.

$\mathrm{Law}^{\mathrm{op}}$ embeds into $\widehat{\mathrm{Law}^{\mathrm{op}}}$ via the Yoneda embedding: for each Lawvere theory L there is a presheaf $y(L)$ in $\widehat{\mathrm{Law}^{\mathrm{op}}}$ such that:

$$y(L)(L') = \mathrm{Hom}_{\mathrm{Law}}(L, L')$$
$$y(L)(l') = \lambda f.\, l' \circ f$$

For an operation $op : P \rightarrowtail A$ of a Lawvere theory L and some presheaf P, we can build a morphism:

$$op_{\mathcal{T}P} : \mathrm{Hom}_{\widehat{\mathrm{Law}^{\mathrm{op}}}}(y(L) \times \Box P \times (\mathcal{T}P)^{\Box A}, \mathcal{T}P)$$
$$(op_{\mathcal{T}P})_{L'} = \lambda(y, p, k).(y(op))_{T_{L'}}(p, \lambda a.k_{L'}(id_{L'}, a))$$

which uses an element of $y(L)$ to map op into the ambient monad and then uses that to construct a term in the free algebras monad.

We use $y(L)$ as the semantics for our handler types, and $op_{\mathcal{T}P}$ as the semantics for reflection.

$$\begin{aligned}
&[\![y(\Sigma)]\!] = y([\![\Sigma]\!]) \\
&[\![\Gamma \vdash_C \Uparrow (v_1(op))(v_2, x.c) : \tau]\!] = \\
&\quad op_{\mathcal{T}[\![\tau]\!]} \circ \langle [\![\Gamma \vdash_V v_1 : y(\Sigma)]\!], [\![\Gamma \vdash_V v_2 : \tau_p]\!], \lambda([\![\Gamma; x : \tau_a \vdash_C c : \tau]\!]) \rangle
\end{aligned}$$

where $[\![\Sigma]\!]$ is the Lawvere theory corresponding to the signature Σ. If Σ is:

$$\{\, op_1 : \tau_{p_1} \rightarrowtail \tau_{a_1} \,;\, \ldots \,;\, op_n : \tau_{p_n} \rightarrowtail \tau_{a_n} \,\}$$

then $[\![\Sigma]\!]$ is the Lawvere theory for:

$$\{\, op_1 : [\![\tau_{p_1}]\!](\bot) \rightarrowtail [\![\tau_{a_1}]\!](\bot) \,;\, \ldots \,;\, op_n : [\![\tau_{p_n}]\!](\bot) \rightarrowtail [\![\tau_{a_n}]\!](\bot) \,\}$$

Note that by applying the types to the initial theory $\bot$ we restrict them to their global elements: those that do not depend on the current ambient monad.

The Yoneda embedding preserves finite products, which for our case means:

$$y(L + L') \cong y(L) \times y(L')$$

which we use for the semantics of out_L and out_R.

4.7 Representing effectful computations: The Yoneda lemma

We can represent computations in a specific algebraic monad using the Yoneda lemma. The Yoneda lemma says that, for any $P : \widehat{\mathbb{C}}$ and $C : \mathbb{C}$, there is an isomorphism:

$$\text{Hom}_{\widehat{\mathbb{C}}}(y(C), P) \cong P(C)$$

Applying that to the composition of the free algebras monad and the global modality we get:

$$\text{Hom}_{\widehat{\text{Law}^{\text{op}}}}(y(L), \mathcal{T} \Box \tau) \cong \mathcal{T} \Box \tau(L) = T_L(\tau(\bot))$$

In terms of our semantics, this means that computations of type $\Box\, \tau$ with a single free variable of type $y(\Sigma)$ are isomorphic to values of the free algebra of $[\![\Sigma]\!]$ at $[\![\tau]\!](\bot)$.

4.8 Effect reification and inductive types

The essence of algebraic effect handlers is taking a computation in $T_{L+\Sigma}(S)$ the free algebras of the sum of an algebraic theory L and an algebraic signature Σ, splitting out and handling Σ, leaving a computation in only the algebraic theory $T_L(S')$.

The key to this process is the following lemma [8].

Lemma 4.7. *For any Lawvere theory L and algebraic signature Σ*

$$T_{L+\Sigma}(S) \cong \mu X.T_L(S + F_\Sigma(X))$$

where $\mu X.G(X)$ is the initial algebra of some polynomial endofunctor G. This lemma is crucial because it shows that computations using both ambient effects L and the signature Σ can be reorganized as a recursive structure that separates the two effect sources – exactly what effect handlers need.

Effect reification works by exposing $\mu X.T_L(S+F_\Sigma(X))$ to the user via an inductive type. Categories of presheaves have initial algebras of polynomial endofunctors, i.e. W-types, so we can construct the required initial algebra as a presheaf. Given an algebraic signature Σ and a presheaf P, we define the following family of initial algebras:

$$\begin{aligned}
&W_\Sigma : [\widehat{\text{Law}^{\text{op}}}, \widehat{\text{Law}^{\text{op}}}] \\
&W_\Sigma(P) = \mu X.(P + F_\Sigma \circ \mathcal{T} X)
\end{aligned}$$

and observe the following variation on Lemma 4.7.

Lemma 4.8. *For any Lawvere theory L and algebraic signature Σ*

$$T_{L+\Sigma}(P(L)) \cong \mathcal{T}(W_\Sigma(P))(L)$$

We use these initial algebras as the denotational semantics of W_Σ:

$$[\![W_\Sigma(\tau)]\!] = W_{[\![\Sigma]\!]}([\![\tau]\!])$$

Exponentials in categories of presheaves are defined such that:

$$Q^P(C) \cong \mathrm{Hom}_{\widehat{\mathbb{C}}}(y(C) \times P, Q)$$

Applying this to $\mathcal{T}(\Box P)^{y(L)}$ and then applying the Yoneda Lemma we get:

$$\begin{aligned}
\mathcal{T}(\Box P)^{y(L)}(L') &\cong \mathrm{Hom}_{\widehat{\mathrm{Law}^{\mathrm{op}}}}(y(L') \times y(L), \mathcal{T}(\Box P)) \\
&\cong \mathrm{Hom}_{\widehat{\mathrm{Law}^{\mathrm{op}}}}(y(L' + L), \mathcal{T}(\Box P)) \\
&\cong \mathcal{T}(\Box P)(L' + L) \\
&\cong T_{(L'+L)}(P(L'))
\end{aligned}$$

In the case where L is a signature Σ, we can further apply Lemma 4.8 and, observing that all the steps in this proof were natural in L', produce this isomorphism:

$$\mathcal{T}(\Box P)^{y(\Sigma)} \cong \mathcal{T}(W_\Sigma(\Box P))$$

which we call $\Downarrow_\Sigma$.

$\Downarrow_\Sigma$ gives us our semantics for reification.

$$[\![\Gamma \vdash_C \Downarrow_\Sigma(x.\, c) : W_\Sigma(\Box\, \tau)]\!] = \Downarrow_{\Sigma\tau} \circ \lambda([\![\Gamma;\, x : y(\Sigma) \vdash_C c : \Box\, \tau]\!])$$

5 Implementation

5.1 Implementation with untracked effect handlers

We can implement the interface from Section 2.2 directly in terms of OCaml's untracked effect handlers [15][2]. The trick is to generate a fresh effect constructor for each reification, pass that constructor around as part of the `handler`, and then use it to perform operations during reflection. The full implementation is given in Fig.8.

5.2 More efficient implementation in terms of fibers

In the implementation in Fig.8, whenever an operation is reflected we perform a linear search up the stack of handlers until we find the corresponding one. This search is not actually necessary: there is a one-to-one relationship between the handlers and the handler values used to reflect operations. The linear search could be eliminated by extending the OxCaml runtime to support direct references to handlers. By allowing local values to point directly to the fiber structures that implement effect handlers, operations can jump immediately to the correct handler without traversing the stack. This optimization maintains the same interface while providing constant-time operation dispatch. It is roughly the implementation technique described by Ma et al. [12].

[2]For convenience, we use the new handler syntax from OCaml 5.3. However, the most recently released version of OxCaml is based on OCaml 5.2 so the actual implementation uses `Effect.Deep.match_with` instead

```
module Handler (O : Op) = struct

  type t = { perform : 'r. 'r O.t -> 'r }

end

module Reflection (O : Op) = struct
  open Term(O)
  open Handler(O)

  let reify f =
    let module Eff =
      struct type 'a Effect.t += C : 'a O.t -> 'a Effect.t end
    in
    let handler =
      { perform = fun o -> Effect.perform (Eff.C o) }
    in
    match f handler with
    | v -> Return v
    | effect Eff.C o, k -> Op(o, fun x -> Effect.continue k x)

  let reify_local f = exclave
    let module Eff =
      struct type 'a Effect.t += C : 'a O.t -> 'a Effect.t end
    in
    let handler = { perform = fun o -> Effect.perform (Eff.C o) } in
    match f handler with
    | v -> Return v
    | effect Eff.C o, k -> Op(o, fun x -> Effect.continue k x)

  let perform t =
   fun h -> h.perform t

end

module Project (L : Op) (R : Op) = struct
  open Sum(L)(R)

  let outl ({perform} : Handler(Sum(L)(R)).t) : Handler(L).t =
    exclave {perform = fun op -> perform (Left op)}

  let outr ({perform} : Handler(Sum(L)(R)).t) : Handler(R).t =
    exclave {perform = fun op -> perform (Right op)}

end
```

Fig. 8. Implementation

6 Related work

6.1 Lexical and named effect handlers

Effect reflection is closely related to *lexical effect handlers* in systems such as Effekt [4] or Biernacki et al. [2]. Lexical effect handlers have a value that represents the effect handler, similar to $y(\Sigma)$ values, but these values are second class and require their own special typing rules. More recent work on Effekt [3] addresses the second-class nature of these values, but requires further extending the type system by tracking the set of capabilities used by each value.

Effect reflection is also closely related to *named effect handlers* [17]. Named effect handlers use higher-rank polymorphism to emulate locality tracking in much the same way as the `St` monad in Haskell. This still requires using an effect system to track which named effects are used by a computation, in order to ensure that those effects do not escape the scope of their handler. Using higher-rank polymorphism also prevents global type inference, requiring complex inference that extends beyond classic Hindley-Milner, and demanding a higher annotation burden on the programmer.

6.2 Monadic reflection

Effect reflection is a variation on monadic reflection [6]. By restricting ourselves to algebraic monads with no equations we remove the need to manually define *glue* functions that describe the interaction between the monad being reified and the current ambient monad. By using locality and $y(\Sigma)$ values we avoid the need to have an effect system.

6.3 Semantics of algebraic effect handlers

Hyland et al. [8] investigate combining algebraic effects, including the key property of sums of algebraic theories (our Lemma 4.7) on which handlers are based.

The original work on algebraic effect handlers [13, 14] gives a semantics in terms of free algebras but the semantics work with a fixed algebraic theory: handlers do not actually change the ambient monad. Our presheaf semantics naturally accommodates changing ambient monads.

6.4 Comonadic purity

Our use of a comonadic global modality to track values which do not depend on the ambient monad is essentially the same as the purity comonad described by Choudhury and Krishnaswami [5]. Their semantic description of their comonad is very different – based on a semantics of capabilities – but the typing rules, and the practical result, are the same.

6.5 Locality and presheaves

Our presheaf semantics are closely related to presheaf semantics for higher-order abstract syntax [7]. In both cases presheaves are used to index the semantics of types and terms by some notion of context, with the Yoneda embedding used to access this context and the global modality used to control dependence on it.

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, pages 13–24, 2013.

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[3] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–30, 2022.

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[5] Vikraman Choudhury and Neel Krishnaswami. Recovering purity with comonads and capabilities. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–28, 2020.

[6] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 175–188, 1999.

[7] Jason ZS Hu, Brigitte Pientka, and Ulrich Schöpp. A category theoretic view of contextual types: From simple types to dependent types. *ACM Transactions on Computational Logic*, 23(4):1–36, 2022.

[8] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical computer science*, 357(1-3):70–99, 2006.

[9] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *ACM SIGPLAN Notices*, 50(12):94–105, 2015.

[10] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 486–499, 2017.

[11] Anton Lorenzen, Leo White, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. Oxidizing OCaml with modal memory management. *Proceedings of the ACM on Programming Languages*, 8(ICFP):485–514, 2024.

[12] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. Lexical effect handlers, directly. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1670–1698, 2024.

[13] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[14] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical methods in computer science*, 9, 2013.

[15] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 206–221. ACM, 2021.

[16] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.

[17] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. First-class names for effect handlers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):30–59, 2022.