Algebraizing Higher-Order Effects

Martin Andrieux, Alan Schmitt

Algebraizing Higher-Order Effects

Martin Andrieux[1] and Alan Schmitt[2]

[1]Université de Rennes
[2]INRIA

We present a technique for expressing higher-order effects as algebraic ones. Algebraic effects provide a clean and modular account of computational effects, but they exclude higher-order effects such as `local` or `catch`. The standard representation of higher-order effects breaks the separation between syntax and interpretation that algebraic effects rely on. Our proposal, effect algebraization, transforms higher-order effects into combinations of algebraic effects. Each higher-order effect is split into a pair of operations—*Open* and *Close*—that together capture the same behavior using only algebraic constructs. The approach is illustrated on *reader* effects, `ask` and `local`. We also sketch a proof of semantics preservation: for every interpretation of the original higher-order effects, there exists a corresponding interpretation of the algebraized ones yielding the same result.

1 Introduction

Algebraic effects and handlers have become a popular foundation for modeling computational effects, thanks to their modular and compositional nature. In this setting, effects are defined abstractly, as operations whose meaning is later specified by handlers. This separation of syntax (what effects are available) from semantics (how they are interpreted) simplifies the abstract reasoning about programs and the combination of multiple effects.

However, this clean separation breaks down when considering higher-order effects, i.e., effects whose semantics depends on a subcomputation. These include local environments in readers, exception catching, or resource management constructs such as context managers[1]. Such effects are usually implemented as handlers of algebraic effects that delimit a region of interpretation. While this approach is expressive, it ties the effect's meaning to a particular interpretation, thereby losing modularity. Once an effect is implemented as a handler, its semantics is fixed within that scope and cannot be reinterpreted or composed freely. This limitation becomes problematic when one wants to reason about or transform effectful computations at the syntactic level, such as during compilation or semantic analysis.

Our broader goal is to derive compilers automatically from semantic descriptions expressed as effectful evaluators (in particular, skeletal semantics [BGJS19, NS22]). Our short-term objective is to build a shallow embedding of skeletal semantics in the Rocq proof assistant [BB22, ALS22, XZH+19]. For this purpose, we need to manipulate the syntax of effectful computations while keeping the effects distinct from their interpretations.

As long as all effects are algebraic, this separation is straightforward: transformations such as continuation-passing style (CPS) or defunctionalization can be applied directly to

[1]`https://docs.python.org/3/reference/datamodel.html#context-managers`

the syntactic representation of effects. But higher-order effects break this uniformity, since their semantics are encoded in handlers rather than in the syntax of operations themselves.

To address this issue, we propose a general technique called *effect algebraization*. The key idea is to transform higher-order effects into combinations of purely algebraic operations. Each higher-order effect is replaced by a pair of algebraic ones—*Open* and *Close*—that together capture its behavior in an entirely syntactic way. This transformation restores the separation between syntax and semantics, allowing higher-order effects to be expressed, combined, and manipulated using the same algebraic machinery as first-order effects. In particular, it enables us to treat scoped effects as algebraic, making them compatible with transformations such as CPS translation and abstract machine derivation.

The paper is structured as follows. Section 2 introduces the modularity problem. Section 3 illustrate the *effect algebraization* procedure, which is the main contribution of this work. Section 4 establishes semantics preservation, an accompanying Agda file can be found at `https://skeletons.inria.fr/papers/Algebraization.agda`. Finally, Section 5 discusses related work.

2 Background

Effects are abstract operations used to model *side-effects* of programming languages. By abstract, we mean that the definition (or *syntax*) of such an operation is disjoint from its meaning (or *semantics*). For example, a `flip :: NDet Bool` operation of type "non-deterministic boolean" could return a truely random value. It could also pick its result from a predefined list of outputs or even always return `True` while keeping the same interface (i.e., the same type). This separation of concerns enables multiple interpretations of effects and the definition of custom *interactions* between effects. For instance, when using non-determinism and state, one can choose to either keep or discard the changes done in the memory when a branch fails by changing how the effect is handled, without changing the code itself.

In addition, effect systems track in the type system which effects a computation may perform. A function type such as `Reader r a` signals that the computation may access a read-only environment of type `r`. Other effects, such as exceptions or non-determinism, can be described in the same way with types like `Except e a` or `NDet a`. Programming languages rely on effects to interact with the outside world and to structure control flow.

Over the years, several frameworks have been proposed to describe effects and their composition. They differ in style, in expressive power, and in modularity. We begin by reviewing monad transformers in Section 2.1, the earliest and most widely used approach. We then turn to algebraic effects and handlers in Section 2.2, and finally to free monads and their higher-order generalization in Section 2.3.

2.1 Monad Transformers

Monad transformers are stackable type constructors that, when combined, form a monad. Each transformer represents a family of effects—such as reading or writing state, or throwing and catching exceptions—while the order of transformers in the stack determines how these effects interact. Modularity is achieved through type classes, which abstract over the position of a transformer in the stack. When adding a transformer `t` to a stack `m a`, one must specify how the existing effects interact with the new one. This typically requires defining a *lift* function: `lift :: (Transformer t) => m a -> t m a`.

However, writing such a `lift` function is not always straightforward. As a result, certain stacks do not provide all the effects of the individual transformers they contain, simply because a suitable lifting function is missing. For example, Haskell's MTL[2] type `ContT r (Writer w)`—which stacks a continuation transformer above a write-only state monad—fails

[2]Monad Transformer Library: `https://github.com/haskell/mtl`

to provide the *writer* effects. This *lifting problem* is a well-known issue with the continuation transformer. A similar issue arises when stacking `ContT r` above the exception monad.

The `ContT` type definition in the MTL library even notes (source file is available at `https://hackage-content.haskell.org/package/transformers-0.6.2.0`): "`ContT` is not a functor on the category of monads, and many operations cannot be lifted through it."

In categorical terms, there is no general way to transport a monad morphism (a natural transformation that preserves monadic laws) through `ContT`, as illustrated in Figure 1, where C represents the `ContT r` mapping.

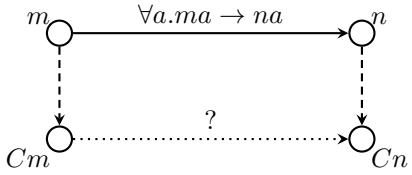


Figure 1. Illustration of the Lack of Monad Morphism after Applying `ContT`

In practical terms, this limitation prevents us from lifting certain effects. For instance, the `local :: env -> Reader env a -> Reader env a` effect is a monad morphism from `Reader env` to itself. Hence, there is no general way to lift it to the transformed monad `ContT r (Reader env) a -> ContT r (Reader env) a`.

This limitation is particularly relevant when deriving abstract machines. The standard transformation from evaluator to abstract machine [ABDM03, ADM03] relies on CPS translation, which corresponds exactly to stacking the `ContT` transformer at the top of the effect stack. While a workaround exists for the `local` effect—using an additional call to `local` to restore the environment before invoking the continuation—it is a special case that does not generalize to other effects.

2.2 Algebraic Effects and Handlers

Algebraic effects and their handlers are a more recent approach to structuring effects [PP13]. The idea is simple: first, we declare effects, which are operations that can be *performed*, much like calling a function. Then, separately, we provide *handlers* that give meaning to these effects. Handlers can interpret multiple effects and can themselves be composed, similar to stacking monad transformers. One handler can forward unhandled effects to handlers further in the stack, providing modularity and flexibility in defining program semantics.

However, not all effects fit neatly into this framework. Algebraic effects are characterized by a property called *algebraicity*, which expresses that effects commute with sequencing. Formally, if `op` is an effect and `>>=` is the monadic *bind*, the following equality holds:

$$\texttt{op}(e, p \to s) \texttt{ >>= } \kappa = \texttt{op}(e,\ p \to s \texttt{ >>= } \kappa)$$

Intuitively, this law says that the continuation κ can always be "pushed inside" the effect without changing the result. The `flip` effect satisfies this law, hence it is algebraic. The `local` effect—which temporarily changes a read-only environment for a subcomputation—is *not* algebraic. To illustrate this idea, we consider the `triple` computation defined in Listing 1.

The type of the computation tells that `triple` performs *read* effects on an integer environment; and that the computation returns an integer. First, `triple` asks for the local environment and remembers its value in a variable `r`. Then, it *locally* updates the environment with the value `2 * r` and runs the subcomputation `ask`, returning this value and remembering it as `x`. Finally, it reads the environment a third time and stores it in the `y` variable. At this point, the code is not affected by the `local` effect anymore, so `y` and `r` are equal. The value returned by the computation is `x + y = (2 * r) + r = 3 * r`, hence the name of the function.

```
triple :: Reader Int Int
triple = do
  r <- ask
  x <- local (2 * r) ask
  y <- ask
  return (x + y)
```

Listing 1. The `local` effect in action, the second `ask` must return `2 * r`

In this example, `ask` behaves differently depending on whether it is inside or outside the scope of `local`. This breaks the algebraicity property: continuations cannot be merged *inside* `local`.

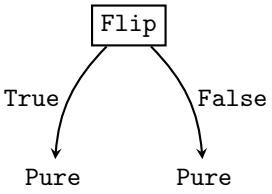
$$\texttt{op}(\texttt{local}_r, \texttt{()} \to \texttt{return ()}) \texttt{ >>= ask} \neq \texttt{op}(\texttt{local}_r, \texttt{ () } \to \texttt{ask})$$

We say that `local` is a *higher-order* effect. Higher-order effects are commonly represented as *handlers of algebraic effects*, considering `local` as an *interpretation* of the `ask` effects in a sub-computation. The downside of this approach is that the semantics of higher-order effect are fixed by the handlers: they no longer are unspecified effects. Unlike algebraic effects, where syntax and semantics remain fully separate, higher-order effects blur this distinction, tying the operation more tightly to a particular interpretation.

2.3 Free Monad and Higher-Order Generalization

In order to give a better formalization of the problem, we introduce the *free monad* structure. This model serves as a foundation for algebraic effects and their handlers. We show how algebraic effects can be encoded in the pure framework of free monads and how free monads can be extended to support higher-order effects.

Intuitively, a *free monad* represents a computation as a tree of operations, illustrated on the right. The inner nodes correspond to effect calls (here a `Flip` effect flipping a coin), the edges to return values of the effect (here `True` or `False`), and the leaves to *pure* values. One definition of the `Free` monad is given below, in the style of Kiselyov's *free-er monad*. The type is parameterized by the effects signatures `eff`.




```
data Free (eff :: Type -> Type) (a :: Type) where
  Pure :: a -> Free eff a
  Op   :: eff b -> (b -> Free eff a) -> Free eff a
```

In this type, each `Op` operation represents an effect of type `eff b`, where `b` is the "return type" of the effect. The *children* of a node are represented by a function from `b` to trees—if `b` is the `Bool` type, a node has two children.

The monad instance of the `Free` monad is straightforward.

```
instance Monad (Free eff) where
  return = Pure
  (Pure x) >>= f = f x
  (Op op k) >>= f = Op op (k >=> f)
```

The definition of the *bind* function enforces the algebraicity property: continuations are merged inside the effect's continuation.[3]

[3]The fish operator `>=>` is the composition of Kleisli arrows, where `f >=> g` is `\x-> f x >>= g`.

In order to build a computation within the free monad, one must start by writing an `Eff` type, telling *what* operations are available to the computation. For instance, the `FlipS` type provides a single operation `Flip` which returns a boolean value. The helper function `flip` represents a computation performing a single `Flip` operation and returning its result. In the graphical depiction of `flip` above, the `Pure` leaves denotes places where the continuation of the computation is plugged (i.e., how the boolean value is used in the remaining computation).

```
data FlipS a where
  Flip :: FlipS Bool

flip :: Free FlipS Bool
flip = Op Flip Pure
```

A *handler* gives an interpretation of algebraic effects, it takes the form of a function `Free eff a -> b`. In the case of the `flip` effect, we can define multiple handlers corresponding to various interpretations of `Flip`. The `handleTrue` handler always returns the `True` boolean value to the continuation, `handlePseudo` achieves pseudo-random number generation with a state monad, and `handleRandom` uses true randomness inside the `IO` monad.

```
handleTrue :: Free FlipS a -> a
handleTrue (Pure x) = x
handleTrue (Op Flip k) = handleTrue (k True)

handlePseudo :: Free FlipS a -> Seed -> a
handlePseudo (Pure x) _ = x
handlePseudo (Op Flip k) seed = handlePseudo (k (bool seed)) (next seed)

handleRandom :: Free FlipS a -> IO a
handleRandom (Pure x) = return x
handleRandom (Op Flip k) = coinFlip >>= handleRandom . k
```

Just like handlers of algebraic effects, free monads are unable to represent higher-order effects without committing to a particular interpretation. In concrete terms, the `triple` computation introduced in the previous section cannot be represented as a computation of type `Free eff Int` without giving an interpretation of the local `ask`.

In recent works [YPW+22, vdBS24], van den Berg & Schrijvers generalize the free monad to support modular handling of higher-order effects. Such representations generalize types with the following shape:

```
data Reader (r :: Type) (a :: Type) where
  Pure :: a -> Reader r a
  Ask :: (r -> Reader r a) -> Reader r a
  Local :: r -> Reader r b -> (b -> Reader r a) -> Reader r a
```

This type is close to the `Free` monad, but the `Local` constructor explicitly represent the subcomputation of type `Reader r b`. This inner computation prevents the `Reader r` type to be expressed as `Free Eff`, because `Eff` would have to reference the `Reader r` type itself. Such a recursion is forbidden in Plotkin and Pretnar's framework [PP13], where `Eff` can only refer to *base* types.

The generalization of the free monad allows for such a recursion. Effect signatures are extended to accept the monad itself as argument, allowing sub-computations to be nested inside effects. We give the definition of the free monad for higher-order effects in Listing 2. In the same listing, we give the signature of the reader effects `Ask` and `Local` to illustrate how to use the higher-order feature: `Local` can refer to the type of computations as `f`.

This additional machinery makes handling of effects harder to reason about. The interpretation must be applied inside effects themselves, merging effects and programs in an unsatisfying way. In order to reduce the complexity of higher-order operations, we propose to explore a new way of dealing with higher-order effects. Instead of extending the free monad to support more effect, we propose to bring higher-order effects back to the algebraic world. We call this procedure *effect algebraization*.

```
data HOFree (eff :: (Type -> Type) -> Type -> Type) (a :: Type) where
  Pure :: a -> HOFree eff a
  Op   :: eff (HOFree eff) b -> (b -> HOFree eff a) -> HOFree eff a

data HOEff f a where
  Ask   :: HOEff f r
  Local :: r -> f b -> HOEff f b
```

Listing 2. Free Monad for Higher-Order Effects

3 Effect Algebraization

Higher-order effects such as `local` and `catch` lie outside the algebraic framework. Effect algebraization is a technique we introduce to rewrite such effects into combinations of algebraic ones. Once this transformation is done, the resulting system can be handled using the standard machinery of algebraic effects.

3.1 The Basic Idea

We recall the `Reader` type defined in the previous section.

```
data Reader (r :: Type) (a :: Type) where
  Pure :: a -> Reader r a
  Ask :: (r -> Reader r a) -> Reader r a
  Local :: r -> Reader r b -> (b -> Reader r a) -> Reader r a
```

We previously explained that this type cannot be expressed in the free monad framework because of the nested computation `Reader r b`. Instead of extending the free monad to support recursion with effects, we propose to change the effects signature in order to make it compatible with the free monad (i.e., transform effects so they become algebraic).

The operation `Ask` is already an algebraic effect, for multiple reasons. First, it satisfies the algebraicity property. Additionally, its implementation in the `HOEff` type (Listing 2) does not use the `f` parameter to create sub-computations, fitting in the well known shape of algebraic effects signatures (in the `Free` type of Section 2.3).

The `Local` operation is the crux of the problem, as it contains two continuations: an *inner* continuation of type `Reader r b`, and an *outer* continuation of type `b -> Reader r a`. This mismatch prevents `Local` from being expressed algebraically. The idea behind effect algebraization is simple: every computation contained in an effect should ultimately return a value of the same type `a`. We can then merge them into a single continuation with a well known type isomorphism `merge :: (b -> a) -> (c -> a) -> ((Either b c) -> a)`.

To achieve this for `Local`, we must transform the inner computation of type `Reader r b` to one of type `Reader r a`. In other words: we must append to it a continuation of type `forall a . b -> Reader r a`. This is a *failure* continuation that consumes a value of type `b` and never returns. This exception can be modeled with a *Close* effect throwing a value of type `b` to the runtime while discarding the continuation. As *Close* is not part of the effects provided by `Reader r`, we must add it the set of effect signatures.

The purely algebraic version of the reader monad is given below. The type of the inner computation has been modified to `a` and the new effect `Close` is added to the signature.

```
data AReader (r :: Type) (a :: Type) where
  Pure :: a -> AReader r a
  Ask :: (r -> AReader r a) -> AReader r a
  Local :: r -> AReader r a -> (b -> AReader r a) -> AReader r a
  Close :: b -> AReader r a
```

The resulting `AReader` type can be expressed as a free monad, with the following `ReaderS` effects signature. In `ReaderS r b`, the type `r` stands for the type of environments and the

type `b` for answers, i.e., the input type of the continuation. The `Void` type is empty, hence it corresponds to discarding the continuation. One can verify that the two definitions of the `AReader` type are equivalent (the `merge` function is needed). Equivalence with the `Reader` type—capable of performing higher-order effects—is discussed in Section 4.

```
data ReaderS r b where
  Ask :: ReaderS r r
  OpenLocal :: r -> ReaderS r (Either () b)
  Close :: b -> ReaderS r Void

type AReader r a = Free (ReaderS r) a
```

A central idea behind this transformation is the annihilation of the continuation performed by the close effect. In the listing below, we give the algebraized version of the `triple` function depicted in Listing 1, denoted `tripleAlg`. Adding a continuation `k : Int -> AReader r out` to `tripleAlg` gives the `tripleThen` function. As the `close` effects discards the continuation, we have `close >>= k` = `close`. The elimination of `k` in the inner computation ensures that outer `ask` operations are not captured by the `local` effect.

```
tripleAlg :: Free AEff Int
tripleAlg = do
  r <- ask
  c <- openLocal (2 * r)
  case c of
    Left () -> -- inner continuation
      ask >>= close
    Right x -> -- outer continuation
      ask >>= \y -> return (x + y)
```

```
tripleThen :: AReader r out
tripleThen = do
  r <- ask
  c <- openLocal (2 * r)
  case c of
    Left () -> -- inner continuation
      ask >>= close >>= k
    Right x -> -- outer continuation
      ask >>= \y -> k (x + y)
```

This elimination is automatic and enforced by the return type of the `close` effect. When writing a code transformation (such as CPS), we consider `openLocal` and `close` as two standard algebraic operations without any distinction with more common effects such as `ask`.

To summarize this section, we give the four steps of the effect algebraization procedure. This steps scale to various effects such as exception handling, resource management, or listening write-only state. We illustrate the procedure with exception handling.

1. Pick a higher-order effect	`catch :: M b -> (e -> M b) -> M b`
2. Explicit the outer continuation [RJ17]	`M b -> (e -> M b) -> (b -> M a) -> M a`
3. Uniformize the types (using `close`)	`M a -> (e -> M a) -> (b -> M a) -> M a`
4. Merge the continuations	`((Either3 () e b) -> M a) -> M a`
5. Build the *Open* effect	`OpenCatch :: Eff (Either3 () e b)`

3.2 Interpreting Algebraized Effects

In this section, we show that an algebraized effect can be interpreted in a modular way. In Section 4, we show that any interpretation of the higher-order effects can be used to interpret the algebraized effects.

As we now live in the algebraic world, we must define the semantics of algebraized effect using a handler—just like we did with the `Flip` effect in Section 2.3. The main challenge of handlers is to link each *Close* to its *Open*, as a value of an unknown type `b` is passed from one to the other. This information can be encoded in the type system (see the Agda development given in Section 4). However, to keep things simple and familiar to the Haskell user, we give a pseudo-Haskell interpreter (which does not typecheck). The interpreter uses a *stack* in order to represent the outer continuation with the former values of the environment.

```
handleStack :: Free (ReaderS r) a -> r -> [(r, b -> Free (ReaderS r) a)] -> a
handleStack (Pure x) _ [] = x -- typing guarantees the stack is empty
handleStack (Op Ask k) r stack = handleStack (k r) r stack
handleStack (Op (OpenLocal new) k) r stack =
  handleStack (k (Left ())) new ((r, k . Right):stack)
handleStack (Op (Close b) _) _ ((r, k):ks) = handleStack (k b) r ks
```

Effect algebraization is not free and the inherent complexity of higher-order effects has to be handled at some point. It is the goal of the handler to reorganize the different parts of computation to recover the original semantics. Most of the time, this requires to store continuations so they can be resumed later (i.e., when the corresponding `close` is performed).

With a defunctionalization step, it becomes possible to build a table containing the effects continuations, allowing to store an index instead of a whole continuation. This approach intuitively builds *basic blocks*, among which the interpreter jumps, storing the address of the next block to execute in a runtime stack. We believe that such an approach will enable efficient compilation of interpreters and automatic generation of compilers from the interpreter.

4 Semantics Preservation

In the previous section, we changed the effects signature and introduced a new effect named `close`. Such an effect-level transformation of computations must be manipulated with care. In particular, we must ensure the following.

> Given an interpretation of the higher-order effects Interp_{HO}, there exists an interpretation of the algebraized effects Interp_{Alg} such that a computation C and its algebraized version $\text{Alg}(C)$ give the same result with their respective interpretation.
>
> $$\forall \text{Interp}_{Ho}\ \exists \text{Interp}_{Alg}\ \forall C,\ \text{Interp}_{HO}(C) = \text{Interp}_{Alg}\big(\text{Alg}(C)\big)$$

In concrete terms, given an interpretation of higher-order effects, we must be able to build an equivalent interpretation of algebraized effects. In particular, we can interpret higher-order effects in the `HOFree` monad itself (by taking the identity function as Interp_{HO}), so we need at least the `rebuild :: Free (Alg eff) a -> HOFree eff a` function. With the `rebuild` function, any interpretation can be used to interpret the algebraized computation (by composing it with `rebuild`). Hence, defining `rebuild` is a necessary and sufficient condition to satisfy the above property.

We define in Agda the algebraization scheme for a monad equipped with mutable state, read-only memory, and exceptions. We successfully write the `rebuild` function and prove the round trip property *rebuild* $\circ$ *algebraize* $=$ *id* for *this particular monad*. The Agda development is available online at the following address: `https://skeletons.inria.fr/papers/Algebraization.agda`.

5 Related Work

We recall the related work that have been described in the body of this paper. Early approaches to structuring computational effects relied on monad transformers, which enable the composition of multiple effects by stacking monads [Mog89, Wad93, LHJ95]. While expressive, they do not provide a notion of "simple" effect and do not support effect-level transformations.

The framework of algebraic effects and handlers—introduced by Plotkin and Pretnar [PP09, PP13]—has gained wide adoption for modeling computational effects. In this setting, effects

are described as abstract operations, and handlers define their interpretations. While transformers and handlers are strongly connected [SPWJ19], the separation between syntax and semantics enables modularity and compositional reasoning [SWI08]. However, algebraic effects are limited to first-order operations that satisfy the algebraicity law—meaning the continuation of a computation can always be pushed inside the effect. This excludes higher-order effects such as `local` (from the reader monad) or `catch`, which depend on subcomputations and thus break algebraicity.

To formally describe higher-order effects while being independent from any interpretation, recent works extend the free monad framework [PSWJ18, YPW+22, vdBS24]. In particular, a generalization of the free monad supports recursive effect signatures, enabling more flexible modeling of subcomputations. However, these extensions blur the line between effects and computations: higher-order effects must be interpreted inside other effects, making reasoning and transformations (such as CPS or defunctionalization [Rey72]) more complex.

Our approach, effect algebraization, differs from these lines of work in spirit and in goal. Instead of extending the free monad to accommodate higher-order effects, we transform them into algebraic operations, thus falling back into the free monad framework. This allows higher-order effects to be expressed, combined, and transformed using standard algebraic machinery, while maintaining modularity.

The idea of expressing higher-order effects with first-order syntax has been studied for scoped effects in [WSH14]. The authors introduce two effects *begin* and *end* in order to delimit scopes. In this same paper, the idea is put aside in favor of higher-order syntax; mainly because of the bracketing problem. More recently, Matache *et al* [MLM+25] introduced parameterized algebraic theories to solve the bracketing problem of algebraic effects. Scopes are resources that are created by *Open*s and consumed by *Close*s. Their framework uses parameterized computations [ATK09] (or *indexed monads*) in order to keep track of the opened scopes. However, the linearity and non-commutativity of scope-resources makes the framework poorly composable, as it only can keep track of a single type of scope.

Our approach shares the main idea of first-order syntax, with applicability and automatic generation in mind. In addition, our *Close* operation discards its continuation, making *Open* the only operation responsible of scope management. Our Agda development (Section 4) uses a parameterized monad to keep track of *Close* operations. We are currently working on the integration of parameterized free monads in order to make interpretation easier and to enforce the structural correctness of computations.

6 Conclusion and Future Work

We have presented effect algebraization, a uniform method for expressing higher-order effects as algebraic ones. The transformation replaces each higher-order effect with a pair of algebraic operations—*Open* and *Close*—whose combined behavior reproduces the original higher-order semantics. The immediate benefit is conceptual: the semantics of an effectful evaluator can now be expressed and transformed entirely within the algebraic setting. Hence, further transformation—like transformation to continuation passing style an defunctionalization—only have to consider algebraic effects. This work is motivated by the need of "interpretation independence" required by our broader objective of syntactically deriving verified compilers from effectful semantic descriptions.

Our current formalization, implemented in Agda, handles a language with mutable state, read-only environments, and exceptions. It includes a proof that algebraization followed by reconstruction yields an equivalent program. Several directions remain open. The first is to formalize the semantics-preservation statement for other algebraic effects—currently, only scoped effects and the `catch` effect are supported. Then, we hope to apply the approach to the automatic derivation of verified abstract machines from skeletal semantics [BGJS19]. Finally, we aim to use equational reasoning over algebraized effects to justify program optimizations and transformations at a syntactic level.

References

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy et Jan Midtgaard : A functional correspondence between evaluators and abstract machines. *In Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, PPDP '03, pages 8–19. Association for Computing Machinery, 2003.

[ADM03] Mads Sig Ager, Olivier Danvy et Jan Midtgaard : A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *BRICS Report Series*, 10(35), Nov. 2003.

[ALS22] Guillaume Ambal, Sergueï Lenglet et Alan Schmitt : Certified abstract machines for skeletal semantics. *In Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, pages 55–67. Association for Computing Machinery, 2022.

[ATK09] ROBERT ATKEY : Parameterised notions of computation. *Journal of Functional Programming*, 19(3–4):335–376, 2009.

[BB22] Maciej Buszka et Dariusz Biernacki : Automating the functional correspondence between higher-order evaluators and abstract machines. *In* Emanuele De Angelis et Wim Vanhoof, éditeurs : *Logic-Based Program Synthesis and Transformation*, pages 38–59. Springer International Publishing, 2022.

[BGJS19] Martin Bodin, Philippa Gardner, Thomas Jensen et Alan Schmitt : Skeletal semantics and their interpretations. *Coq formalisation for Article: Skeletal Semantics and Their Interpretations*, 3:44:1–44:31, 2019.

[LHJ95] Sheng Liang, Paul Hudak et Mark Jones : Monad transformers and modular interpreters. *In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343. Association for Computing Machinery, 1995.

[MLM+25] Cristina Matache, Sam Lindley, Sean Moss, Sam Staton, Nicolas Wu et Zhixuan Yang : Scoped effects, scoped operations, and parameterized algebraic theories. *ACM Trans. Program. Lang. Syst.*, 47(2), juin 2025.

[Mog89] E. Moggi : Computational lambda-calculus and monads. *In [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[NS22] Louis Noizet et Alan Schmitt : Semantics in skel and necro. *In ICTCS 2022 - Italian Conference on Theoretical Computer Science*, page 1, 2022.

[PP09] Gordon Plotkin et Matija Pretnar : Handlers of algebraic effects. *In* Giuseppe Castagna, éditeur : *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[PP13] Gordon Plotkin et Matija Pretnar : Handling algebraic effects. *Logical Methods in Computer Science [electronic only]*, 9, 12 2013.

[PSWJ18] Maciej Piróg, Tom Schrijvers, Nicolas Wu et Mauro Jaskelioff : Syntax and semantics for operations with scopes. *In Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 809–818, New York, NY, USA, 2018. Association for Computing Machinery.

[Rey72] John C. Reynolds : Definitional interpreters for higher-order programming languages. *In Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740. Association for Computing Machinery, 1972.

[RJ17] EXEQUIEL RIVAS et MAURO JASKELIOFF : Notions of computation as monoids. *Journal of Functional Programming*, 27:e21, 2017.

[SPWJ19] Tom SCHRIJVERS, Maciej PIRÓG, Nicolas WU et Mauro JASKELIOFF : Monad transformers and modular algebraic effects: what binds them together. *In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA, 2019. Association for Computing Machinery.

[SWI08] WOUTER SWIERSTRA : Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[vdBS24] Birthe van den BERG et Tom SCHRIJVERS : A framework for higher-order effects & handlers. *Sci. Comput. Program.*, 234(C), mai 2024.

[Wad93] Philip WADLER : Monads for functional programming. *In* Manfred BROY, éditeur : *Program Design Calculi*, pages 233–264. Springer, 1993.

[WSH14] Nicolas WU, Tom SCHRIJVERS et Ralf HINZE : Effect handlers in scope. *In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 1–12, New York, NY, USA, 2014. Association for Computing Machinery.

[XZH+19] Li-yao XIA, Yannick ZAKOWSKI, Paul HE, Chung-Kil HUR, Gregory MALECHA, Benjamin C. PIERCE et Steve ZDANCEWIC : Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), décembre 2019.

[YPW+22] Zhixuan YANG, Marco PAVIOTTI, Nicolas WU, Birthe van den BERG et Tom SCHRIJVERS : Structured handling of scoped effects. *In* Ilya SERGEY, éditeur : *Programming Languages and Systems*, pages 462–491, Cham, 2022. Springer International Publishing.