Structured Handling of Scoped Effects

Zhixuan Yang[1] ✉, Marco Paviotti[1], Nicolas Wu[1], Birthe van den Berg[2], and Tom Schrijvers[2]

[1] Imperial College London, London, United Kingdom
{s.yang20,m.paviotti,n.wu}@imperial.ac.uk
[2] KU Leuven, Leuven, Belgium
{birthe.vandenberg,tom.schrijvers}@kuleuven.be

Abstract. Algebraic effects offer a versatile framework that covers a wide variety of effects. However, the family of operations that delimit scopes are not algebraic and are usually modelled as handlers, thus preventing them from being used freely in conjunction with algebraic operations. Although proposals for scoped operations exist, they are either ad-hoc and unprincipled, or too inconvenient for practical programming. This paper provides the best of both worlds: a theoretically-founded model of scoped effects that is convenient for implementation and reasoning. Our new model is based on an adjunction between a locally finitely presentable category and a category of *functorial algebras*. Using comparison functors between adjunctions, we show that our new model, an existing indexed model, and a third approach that simulates scoped operations in terms of algebraic ones have equal expressivity for handling scoped operations. We consider our new model to be the sweet spot between ease of implementation and structuredness. Additionally, our approach automatically induces fusion laws of handlers of scoped effects, which are useful for reasoning and optimisation.

Keywords: Computational effects · Category theory · Haskell · Algebraic theories · Scoped effects · Handlers · Abstract syntax

1 Introduction

For a long time, monads [45, 60, 68] have been the go-to approach for purely functional modelling of and programming with side effects. However, in recent years an alternative approach, *algebraic effects* [48], is gaining more traction. A big breakthrough has been the introduction of *handlers* [52], which has made algebraic effects suitable for programming and has led to numerous dedicated languages and libraries implementing algebraic effects and handlers. In comparison to monads, algebraic effects provide a more modular approach to computations with effects, in which the syntax and semantics of effects are separated—computations invoking algebraic operations can be defined syntactically, and the semantics of operations are given by handlers separately in possibly many ways.

A disadvantage of algebraic effects is that they are less expressive than monads; not all effects can be easily expressed or composed within their confines.

For instance, operations like *catch* for exception handling, *spawn* for parallel composition of processes, or *once* for restricting nondeterminism are not conventional algebraic operations; instead they delimit a computation within their scope. Such operations are usually modelled as handlers, but the problem is that they cannot be freely used amongst other algebraic operations: when a handler implementing a scoped operation is applied to a computation, the computation is transformed from a syntactic tree of algebraic operations into some semantic model implementing the scoped operation. Consequently, all subsequent operations on the computation can only be given in the particular semantic model rather than as mere syntactic operations, thus nullifying the crucial advantage of modularity when separating syntax and semantics of effects.

To remedy the situation, Wu et al. [70] proposed a practical, but ad-hoc, generalisation of algebraic effects in Haskell that encompasses scoped effects, that has been adopted by several algebraic effects libraries [32, 42, 56]. More recently, Piróg et al. [46] sought to put this ad-hoc approach for scoped effects on the same formal footing as algebraic effects. Their solution resulted in a construction based on a level-indexed category, called *indexed algebras*, as the way to give semantics to scoped effects. However, this formalisation introduces a disparity between syntax and semantics that makes indexed algebras not as structured as the programs they interpret, where they use an ad-hoc hybrid fold that requires indexing for the handlers, but not for the program syntax. Moreover, indexed algebras are not ideal for widespread implementation as they require dependent typing, in at least a limited form like GADTs [25].

This paper presents a more structured way of handling scoped effects, which we call *functorial algebras*. They are principled and formally grounded on category theory, and at the same time more structured than the indexed algebras of Piróg et al. [46], in the sense that the structure of functorial algebras directly follows the abstract syntax of programs with scoped effects. Functorial algebras enjoy the following advantages over indexed algebras:

- Functorial algebras admit a simpler interface and implementation (Figure 1) without requiring dependent types or GADTs. This enables the adoption of scoped effects in a wider range of languages.
- Functorial algebras are easier to reason about due to their structuredness. In particular, it allows us to derive a one-pass handle function (Theorem 2) that does not convert syntax to the free functorial algebra. In comparison, a similar one-pass recursion scheme is much harder for indexed algebras to derive. Although Piróg et al. showed one in their implementation, they did not prove its correctness. In this paper, we provide the missing proof by converting indexed algebras to functorial ones (Example 12).
- These improvements have not sacrificed expressivity, since translating between functorial algebras and existing approaches is possible (Section 4).

The structure and contributions of this paper are as follows:

- We highlight the loss of modularity when modelling scoped operations as handlers and sketch how the problem is solved using functorial algebras in Haskell, along with a number of programming examples (Section 2).

– We develop a category-theoretic foundation of functorial algebras as a notion of handlers of scoped effects. Specifically, we show that there is an adjunction between functorial algebras and a base category, inducing the monad modelling the syntax of scoped effects (Section 3).
– We show that the expressivity of functorial algebras, Piróg et al. [46]'s indexed algebras, and simulating scoped effects with algebraic operations and recursion are equal, by constructing interpretation-preserving functors between the three categories of algebras (Section 4).
– We present the fusion law of functorial algebras, which is useful for reasoning and optimisation. The fusion law directly follows from the naturality of the adjunction underlying functorial algebras (Section 5).

Finally, we discuss related work (Section 6) and conclude (Section 7). An extended version of this paper [71] contains appendices and proofs for this paper.

2 Scoped Effects for the Working Programmer

We start with a recap of *handlers of algebraic effects* (Section 2.1), and then we highlight the loss of modularity when modelling non-algebraic effectful operations as handlers (Section 2.2). We then show how the problem is solved by modelling them as *scoped operations* and handling them with *functorial algebras* in Haskell (Section 2.3), whose categorical foundation will be developed later.

2.1 Handlers of Algebraic Effects

For the purpose of demonstration, in this section we base our discussion on a simplistic implementation of effect handlers in Haskell using *free monads*, although the problem with effect handlers highlighted in this section applies to other more practical implementations of effect handlers, either as libraries (e.g. [27, 33]) or standalone languages (e.g. [7, 36, 40]).

Following Plotkin and Pretnar [52], computational effects, such as exceptions, mutable state, and nondeterminism, are described by *signatures* of primitive effectful operations. Signatures can be abstractly represented by Haskell functors:

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\ \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

The following functor *ES* (with the evident *Functor* instance) is the signature of three operations: throwing an exception, writing and reading an *Int*-state:

$$\textbf{data}\ \mathit{ES}\ x = \mathit{Throw} \mid \mathit{Put}\ \mathit{Int}\ x \mid \mathit{Get}\ (\mathit{Int} \to x) \tag{1}$$

Typically, a constructor of a signature functor Σ has a type isomorphic to $P \to (R \to x) \to \Sigma\ x$ for some types P and R. As in (1), the types of the three constructors are isomorphic to $\mathit{Throw} :: () \to (\mathit{Void} \to x) \to \mathit{ES}\ x$, $\mathit{Put} :: \mathit{Int} \to (() \to x) \to \mathit{ES}\ x$ and $\mathit{Get} :: () \to (\mathit{Int} \to x) \to \mathit{ES}\ x$ respectively where *Void* is the empty type. Each constructor of a signature functor Σ is thought of as an *operation* that takes a parameter of type P and produces a result of type R, or equivalently, has R-many possible ways to continue the computation after the operation. Given any (signature) functor Σ, computations invoking operations from Σ are modelled by the following datatype, called the *free monad* of Σ,

$$\textbf{data}\ \mathit{Free}\ \Sigma\ a = \mathit{Return}\ a \mid \mathit{Call}\ (\Sigma\ (\mathit{Free}\ \Sigma\ a))$$

whose first case represents a computation that just *return*s a value, and the second case represents a computation *call*ing an operation from Σ with more $\mathit{Free}\ \Sigma\ a$ subterms as arguments, which are understood as the continuation of the computation after this call, depending on the outcome of this operation.

The inductive datatype $\mathit{Free}\ \Sigma\ a$ comes with a *recursion principle*:

$$\begin{aligned}
&\mathit{handle} :: (\Sigma\ b \to b) \to (a \to b) \to \mathit{Free}\ \Sigma\ a \to b\\
&\mathit{handle}\ \mathit{alg}\ g\ (\mathit{Return}\ x) = g\ x\\
&\mathit{handle}\ \mathit{alg}\ g\ (\mathit{Call}\ \mathit{op})\ \ = \mathit{alg}\ (\mathit{fmap}\ (\mathit{handle}\ \mathit{alg}\ g)\ \mathit{op})
\end{aligned}$$

which folds a tree of operations $\mathit{Free}\ \Sigma\ a$ into a type b, providing a way $\Sigma\ b \to b$, usually called a *Σ-algebra*, to perform operations from Σ on b and a way $a \to b$ to transform the returned type a of computations to b. The function *handle* can be used to give $\mathit{Free}\ \Sigma$ a monad instance:

$$\begin{array}{ll}
\mathit{return} :: a \to \mathit{Free}\ \Sigma\ a & (\gg\!=) :: \mathit{Free}\ \Sigma\ a \to (a \to \mathit{Free}\ \Sigma\ b) \to \mathit{Free}\ \Sigma\ b\\
\mathit{return} = \mathit{Return} & m \gg\!= k = \mathit{handle}\ \mathit{Call}\ k\ m
\end{array}$$

The monadic instance allows the programmer to build effectful computations using the **do**-notation in a clean way. For example, the following program updates the state s to $n\ /\ s$ for some $n :: \mathit{Int}$, and throws an exception when s is 0:

$$\begin{aligned}
&\mathit{safeDiv} :: \mathit{Int} \to \mathit{Free}\ \mathit{ES}\ \mathit{Int}\\
&\mathit{safeDiv}\ n = \textbf{do}\ s \leftarrow \mathit{get}; \textbf{if}\ s \equiv 0\ \textbf{then}\ \mathit{Call}\ \mathit{Throw}\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else do}\ \{\mathit{put}\ (n\ /\ s); \mathit{return}\ (n\ /\ s)\}
\end{aligned}$$

where the auxiliary wrapper functions (the so-called *smart constructors* in the Haskell community) that invoke *Call* appropriately are

$$\mathit{get} = \mathit{Call}\ (\mathit{Get}\ \mathit{Return}) \qquad \mathit{put}\ n = \mathit{Call}\ (\mathit{Put}\ n\ (\mathit{Return}\ ()))$$

The free monad merely models effectful computations *syntactically* without specifying how these operations are actually implemented. Indeed, the program *safeDiv* above is defined without saying how mutable state and exceptions are implemented at all. To actually give useful semantics to programs built with free monads, the programmer uses the *handle* function above to interpret programs with Σ-algebras, which are called *handlers* in this context.

For example, given a program $r :: \mathit{Free}\ \mathit{ES}\ a$ for some a, a handler $\mathit{catchHdl}\ r :: \mathit{ES}\ (\mathit{Free}\ \mathit{ES}) \to \mathit{Free}\ \mathit{ES}$ that gives the usual semantics to *throw* is

$$\begin{aligned}
&\mathit{catchHdl} :: \mathit{Free}\ \mathit{ES}\ a \to \mathit{ES}\ (\mathit{Free}\ \mathit{ES}\ a) \to \mathit{Free}\ \mathit{ES}\ a\\
&\mathit{catchHdl}\ r\ \mathit{Throw} = r; \qquad \mathit{catchHdl}\ r\ \mathit{op} = \mathit{Call}\ \mathit{op}
\end{aligned} \tag{2}$$

which evaluates r for *r*ecovery in case of throwing an exception, and leaves other operations untouched in the free monad. An important advantage of the approach of effect handlers is that different semantics of a computational effect can be given by different handlers. For example, suppose that in some scenario one would like to interpret exceptions as unrecoverable errors and stop the execution of the program when an exception is raised. Then the following handler can be defined for this behaviour:

$$\begin{array}{l} \mathit{catchHdl}' :: \mathit{Free}\ \mathit{ES}\ a \to \mathit{ES}\ (\mathit{Free}\ \mathit{ES}\ (\mathit{Maybe}\ a)) \to \mathit{Free}\ \mathit{ES}\ (\mathit{Maybe}\ a) \\ \mathit{catchHdl}'\ r\ \mathit{Throw} = \mathit{return}\ \mathit{Nothing}; \quad \mathit{catchHdl}'\ r\ \mathit{op} = \mathit{Call}\ \mathit{op} \end{array} \tag{3}$$

As expected, applying these two handlers to the program *safeDiv* 5 produces different results (of types *Free ES Int* and *Free ES* (*Maybe Int*) respectively):

$$\begin{array}{l} \quad \mathit{handle}\ (\mathit{catchHdl}\ (\mathit{return}\ 42))\ \mathit{return}\ (\mathit{safeDiv}\ 5) \\ = \mathbf{do}\ s \leftarrow \mathit{get}; \mathbf{if}\ s \equiv 0\ \mathbf{then}\ \mathit{return}\ 42\ \mathbf{else\ do}\ \{\mathit{put}\ (n\ /\ s); \mathit{return}\ (n\ /\ s)\} \\ \quad \mathit{handle}\ (\mathit{catchHdl}'\ (\mathit{return}\ 42))\ (\mathit{return} \cdot \mathit{Just})\ (\mathit{safeDiv}\ 5) \\ = \mathbf{do}\ s \leftarrow \mathit{get}; \mathbf{if}\ s \equiv 0\ \mathbf{then}\ \mathit{return}\ \mathit{Nothing} \\ \qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ \{\mathit{put}\ (n\ /\ s); \mathit{return}\ (\mathit{Just}\ (n\ /\ s))\} \end{array}$$

Note that exception *throwing* and *catching* are modelled differently in the approach of algebraic effects and handlers, one as an operation in the signature *ES* and one as a handler, although it is natural to expect both of them to be operations of *the effect of exceptions*. This asymmetry results from the fact that exception catching is *not algebraic*: if *catch* was modelled as a binary operation in the signature, then the monadic bind $\gg\!=$ of the free monad earlier, which intuitively means sequential composition of programs, would imply that $(\mathit{catch}\ r\ p) \gg\!= k = \mathit{catch}\ (r \gg\!= k)\ (p \gg\!= k)$, which is semantically undesirable. Thus the perspective of Plotkin and Pretnar [52] is that non-algebraic operations like *catch* should be deemed different from algebraic operations, and they can be modelled as handlers (of algebraic operations).

2.2 Scoped Operations as Handlers Are Not Modular

However, this treatment of non-algebraic operations leads to a somewhat subtle complication: as observed by Wu et al. [70], when non-algebraic operations (such as *catch*) are modelled with handlers, these handlers play a dual role of (i) modelling the syntax of the operation (the scope for which exceptions are caught by *catch*) and (ii) giving semantics to it (when an exception is caught, run the recovery program). To see the problem more concretely, ideally one would like to have a syntactic operation *catch* of the following type that acts on computations without giving specific semantics a priori,

$$\mathit{catch} :: \mathit{Free}\ \mathit{ES}\ a \to \mathit{Free}\ \mathit{ES}\ a \to \mathit{Free}\ \mathit{ES}\ a$$

allowing to write programs like

$$\mathit{prog} = \mathbf{do}\ \{x \leftarrow \mathit{catch}\ (\mathit{safeDiv}\ 5)\ (\mathit{return}\ 42); \mathit{put}\ (x + 1)\} \tag{4}$$

and the semantics of (both algebraic and non-algebraic) operations in *prog* can be given separately by handlers. Unfortunately, when *catch* is modelled as handlers *catchHdl* or *catchHdl′* as in the last subsection, the program *prog* must be written differently depending on which handler is used:

$$\begin{array}{ll} & \mathbf{do}\ x \leftarrow \mathit{handle}\ (\mathit{catchHdl}\ (\mathit{return}\ 42))\ \mathit{return}\ (\mathit{safeDiv}\ 5); \mathit{put}\ (x+1) \\ \text{vs.} & \mathbf{do}\ \mathit{xMb} \leftarrow \mathit{handle}\ (\mathit{catchHdl}'\ (\mathit{return}\ 42))\ (\mathit{return} \cdot \mathit{Just})\ (\mathit{safeDiv}\ 5) \\ & \quad \mathbf{case}\ \mathit{xMb}\ \mathbf{of}\ \{ \mathit{Nothing} \to \mathit{return}\ \mathit{Nothing} \\ & \qquad\qquad\qquad (\mathit{Just}\ x) \to \mathbf{do}\ r \leftarrow \mathit{put}\ (x+1); \mathit{return}\ (\mathit{Just}\ r) \} \end{array}$$

The issue is that these handlers interpret the operation *catch* in different semantic models, *Free ES a* and *Free ES* (*Maybe a*), and this affects both the value x that is returned, and the way the subsequent *put* is expressed. Therefore, non-algebraic operation *catch* modelled as handlers is not as modular as algebraic operations, weakening the advantage of programming with algebraic effects.

2.3 Scoped Effects and Functorial Algebras

Now we present an overview of a solution to the problem highlighted above by modelling exception catching as *scoped effects* [46] and handle them using *functorial algebras*, which will be more formally developed in later sections.

Syntax of Scoped Operations To achieve modularity for (non-algebraic) operations delimiting scopes, such as *catch*, which are called *scoped operations*, Piróg et al. [46] generalise the free monad *Free* Σ to a monad *Prog* Σ Γ accommodating both algebraic and scoped operations. The monad is parameterised by two functors Σ and Γ, called the *algebraic signature* and the *scoped signature* respectively. The intention is that a constructor $\mathit{Op} :: (R \to x) \to \Sigma\ x$ of the algebraic signature represents an algebraic operation *Op* producing an R-value as usual, whereas a constructor $\mathit{Sc} :: (N \to x) \to \Gamma\ x$ of the scoped signature represents a scoped operation *Sc* creating N-many scopes enclosing programs.

Example 1. As in the previous subsection, the effect of *exceptions* has an algebraic operation for *throwing* exceptions, which produces no values, and a scoped operation for *catching* exceptions, which creates two scopes, one enclosing the program for which exceptions are caught, and the other enclosing the recovery computation. Thus the algebraic and scoped signatures are respectively

$$\mathbf{data}\ \mathit{Throw}\ x = \mathit{Throw} \qquad\qquad \mathbf{data}\ \mathit{Catch}\ x = \mathit{Catch}\ x\ x \tag{5}$$

Example 2. An effect of *explicit nondeterminism* has two algebraic operations for nondeterministic choice and a scoped operation *Once*:

$$\mathbf{data}\ \mathit{Choice}\ x = \mathit{Fail} \mid \mathit{Or}\ x\ x \qquad\qquad \mathbf{data}\ \mathit{Once}\ x = \mathit{Once}\ x \tag{6}$$

The intention is that this effect implements logic programming [20]—solutions to a problem are exhaustively searched: operation *Or p q* splits a search branch into two; *Fail* marks a failed branch; and the scoped operation *Once p* keeps only the first solution found by p, making it *semi-deterministic*, which is useful for speeding up the search with heuristics from the programmer.

Similar to the free monad, the *Prog* monad models the syntax of computations invoking operations from Σ and Γ:

$$\begin{aligned}\textbf{data}\ \mathit{Prog}\ \Sigma\ \Gamma\ a = {} & \mathit{Return}\ a \mid \mathit{Call}\ (\Sigma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)) \\ \mid {} & \mathit{Enter}\ (\Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)))\end{aligned} \tag{7}$$

Thus an element of $\mathit{Prog}\ \Sigma\ \Gamma\ a$ can either (i) *return* an a-value without causing effects, or (ii) *call* an algebraic operation in Σ with more subterms of $\mathit{Prog}\ \Sigma\ \Gamma\ a$ as the continuation after the operation, or (iii) *enter* the scope of a scoped operation. The third case deserves more explanation: the first *Prog* in $(\Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)))$ represents the programs enclosed by the scoped operation, and the second *Prog* represents the continuation of the program after the scoped operation, and thus the boundary between programs inside and outside the scope is kept in the syntax tree, which is necessary because collapsing the boundary might change the meaning of a program. The distinction between algebraic and scoped operations can be seen more clearly from the monadic bind of *Prog* (the monadic return of *Prog* is just *Return*):

$$\begin{aligned}&(\gg\!=) :: \mathit{Prog}\ \Sigma\ \Gamma\ a \to (a \to \mathit{Prog}\ \Sigma\ \Gamma\ b) \to \mathit{Prog}\ \Sigma\ \Gamma\ b \\ &(\mathit{Return}\ a) \gg\!= k = k\ a \\ &(\mathit{Call}\ op) \quad \gg\!= k = \mathit{Call}\ (\mathit{fmap}\ (\gg\!=k)\ op) \\ &(\mathit{Enter}\ sc) \gg\!= k = \mathit{Enter}\ (\mathit{fmap}\ (\mathit{fmap}\ (\gg\!=k))\ sc)\end{aligned}$$

For algebraic operations, extending the continuation $(\gg\!=k)$ directly acts on the argument to the algebraic operation, whereas for scoped operation, $(\gg\!=k)$ acts on the second layer of *Prog*. Thus for an algebraic operation o, $(o\ p) \gg\!= k$ and $o\ (p \gg\!= k)$ have the same representation, whereas for a scoped operation s, $(s\ p) \gg\!= k$ and $s\ (p \gg\!= k)$ have different representations, which is precisely the distinction between algebraic and scoped operations.

The constructors *Call* and *Enter* are clumsy to work with, and for writing programs more naturally, we define *smart constructors* for operations. Generally, for algebraic operations $\mathit{Op} :: F\ x \to \Sigma\ x$ and scoped operations $\mathit{Sc} :: G\ x \to \Gamma\ x$, the smart constructors are

$$\begin{aligned}&op :: F\ (\mathit{Prog}\ \Sigma\ \Gamma\ a) \to \mathit{Prog}\ \Sigma\ \Gamma\ a && sc :: G\ (\mathit{Prog}\ \Sigma\ \Gamma\ a) \to \mathit{Prog}\ \Sigma\ \Gamma\ a \\ &op = \mathit{Call} \cdot \mathit{Op} && sc = \mathit{Enter} \cdot \mathit{fmap}\ (\mathit{fmap}\ \mathit{return}) \cdot \mathit{Sc}\end{aligned}$$

For example, the smart constructor for *Catch* (Example 1) is

$$\begin{aligned}&\mathit{catch} :: \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \to \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \to \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \\ &\mathit{catch}\ h\ r = \mathit{Enter}\ (\mathit{Catch}\ (\mathit{fmap}\ \mathit{return}\ h)\ (\mathit{fmap}\ \mathit{return}\ r))\end{aligned}$$

With all machinery in place, now we can define the program (4) using *Prog* that we could not write with *Free*:

$$\mathit{prog} = \textbf{do}\ \{x \leftarrow \mathit{catch}\ (\mathit{safeDiv}\ 5)\ (\mathit{return}\ 42); \mathit{put}\ (x+1)\}$$

Handlers of Scoped Operations Similar to *Free*, the *Prog* monad merely models the syntax of effectful computations, and more useful semantics need to be given

```
data EndoAlg Σ Γ f = EndoAlg {
  returnE :: ∀x. x → f x,
  callE   :: ∀x. Σ (f x) → f x,
  enterE  :: ∀x. Γ (f (f x)) → f x}
```

```
data BaseAlg Σ Γ f a =
  BaseAlg {callB  :: Σ a → a
          , enterB :: Γ (f a) → a}
```

```
hcata :: (Functor Σ, Functor Γ) ⇒ (EndoAlg Σ Γ f) → Prog Σ Γ a → f a
hcata alg (Return x)    = returnE alg x
hcata alg (Call op)     = (callE alg · fmap (hcata alg)) op
hcata alg (Enter scope) = (enterE alg · fmap (hcata alg · fmap (hcata alg))) scope
handle :: (Functor Σ, Functor Γ)
       ⇒ (EndoAlg Σ Γ x) → (BaseAlg Σ Γ x b) → (a → b) → Prog Σ Γ a → b
handle ealg balg gen (Return x) = gen x
handle ealg balg gen (Call op)  = (callB balg · fmap (handle ealg balg gen)) op
handle ealg balg gen (Enter sc)
  = (enterB balg · fmap (hcata ealg · fmap (handle ealg balg gen))) sc
```

Fig. 1: A Haskell implementation of handling with functorial algebras

by handlers. Although Piróg et al. [46] developed a notion of *indexed algebras* for this purpose, indexed algebras turn out to be more complicated than necessary (we will discuss them in Section 4), and the contribution of this paper is a simpler kind of handlers for scoped operations, which we call *functorial algebras*.

Given signatures Σ and Γ, a functorial algebra for them is a quadruple $\langle f, b, \mathit{ealg}, \mathit{balg}\rangle$ for some functor f called the *endofunctor carrier*, type b called the *base carrier*. The other two components $\mathit{ealg} :: \mathit{EndoAlg}\ \Sigma\ \Gamma\ f$ and $\mathit{balg} :: \mathit{BaseAlg}\ \Sigma\ \Gamma\ f\ b$ are called the *endofunctor algebra* and the *base algebra*. Their types are fully shown in Figure 1. The intuition is that functor f and *ealg* interpret the part of a program enclosed by scoped operations, and the type b and *balg* interpret the part of a program not enclosed by any scopes.

Example 3. The standard semantics of exception catching (cf. handler (2)) can be implemented by a functorial algebra with the conventional *Maybe* functor as the endofunctor carrier with the following *EndoAlg*:

```
excE :: EndoAlg Throw Catch Maybe
excE = EndoAlg {..} where
  returnE = Just
  callE Throw = Nothing
  enterE :: Catch (Maybe (Maybe a))
         → Maybe a
  enterE (Catch Nothing r) = join r
  enterE (Catch (Just k) _) = k
```

For the base carrier that interprets operations not enclosed by any *catch*, a straightforward choice is just taking *Maybe a* as the base carrier for a type a, and setting *callB* = *callE* and *enterB* = *enterE*, which means that operations inside and outside scopes are interpreted in the same way.

In general, we can define a specialised version of *handle* (Figure 1) that only takes an endofunctor algebra as input for interpreting operations inside and outside scopes in the same way:

$$\begin{aligned}&\mathit{handleE} :: (\mathit{EndoAlg}\ \Sigma\ \Gamma\ f) \to \mathit{Prog}\ \Sigma\ \Gamma\ a \to f\ a\\ &\mathit{handleE}\ \mathit{ealg}@(\mathit{EndoAlg}\ \{..\}) = \mathit{handle}\ \mathit{ealg}\ (\mathit{BaseAlg}\ \mathit{callE}\ \mathit{enterE})\ \mathit{returnE}\end{aligned}$$

Applying *handleE excE* to the following program produces *Just* 43 as expected.

$$\mathbf{do}\ \{x \leftarrow \mathit{catch}\ \mathit{throw}\ (\mathit{return}\ 42); \mathit{return}\ (x+1)\} \tag{8}$$

For the non-standard semantics (cf. (3)) that disables exception recovery, one can define another endofunctor algebra excE' by replacing *enterE* in *excE* with

$$\begin{aligned}&\mathit{enterE}' :: \mathit{Catch}\ (\mathit{Maybe}\ (\mathit{Maybe}\ a)) \to \mathit{Maybe}\ a\\ &\mathit{enterE}'\ (\mathit{Catch}\ \mathit{Nothing}\ \_) = \mathit{Nothing}; \quad \mathit{enterE}'\ (\mathit{Catch}\ (\mathit{Just}\ k)\ \_) = k\end{aligned}$$

With excE', handling the program in (8) produces *Nothing* as expected.

Now we provide some intuition for how functorial algebras work. First note that the three fields of *EndoAlg* in Figure 1 precisely correspond to the three cases of *Prog* (7). Thus by replacing the constructors of *Prog* with the corresponding fields of *EndoAlg*, we have a polymorphic function $\mathit{hcata}\ \mathit{ealg} :: \forall x.\ \mathit{Prog}\ \Sigma\ \Gamma\ x \to f\ x$ (Figure 1) turning a program into a value in f.

The function *handle* (Figure 1) takes a functorial algebra, a function $\mathit{gen} :: a \to b$ and a program p as arguments, and it handles all the effectful operations in p by using *hcata ealg* for interpreting the part of p inside scoped operations and *balg* for interpreting the outermost layer of p outside any scoped operations. The function *gen* corresponds to the 'value case' of handlers of algebraic effects, which transforms the a-value returned by a program into the type b for interpretation.

We close this section with some more examples of handling scoped effects with functorial algebras. The supplementary material of this paper also contains an OCaml implementation of functorial algebras and the following examples.

Example 4. The standard way to handle explicit nondeterminism with the semi-deterministic operator *once* (Example 2) is using a functorial algebra with the list functor as the endofunctor carrier together with the following algebra:

$$\begin{array}{ll}\mathit{ndetE} :: \mathit{EndoAlg}\ \mathit{Choice}\ \mathit{Once}\ [\,] & \mathit{enterE} :: \mathit{Once}\ [[a]] \to [a]\\ \mathit{ndetE} = \mathit{EndoAlg}\ \{..\}\ \mathbf{where} & \mathit{enterE}\ (\mathit{Once}\ x) =\\ \quad \mathit{callE} :: \mathit{Choice}\ [a] \to [a] & \quad \mathbf{if}\ x \equiv [\,]\ \mathbf{then}\ [\,]\ \mathbf{else}\ \mathit{head}\ x\\ \quad \mathit{callE}\ \mathit{Fail} \quad\ \ = [\,] & \mathit{returnE} :: a \to [a]\\ \quad \mathit{callE}\ (\mathit{Or}\ x\ y) = x \mathbin{+\!\!+} y & \mathit{returnE}\ x = [x]\end{array}$$

Then applying *handleE ndetE* to the following program produces $[1,2]$ as expected. In comparison, if *once* were algebraic, the result would be $[1]$.

$$\mathbf{do}\ \{n \leftarrow \mathit{once}\ (\mathit{or}\ (\mathit{return}\ 1)\ (\mathit{return}\ 3)); \mathit{or}\ (\mathit{return}\ n)\ (\mathit{return}\ (n+1))\}$$

Example 5. In the last example we used the list functor to interpret explicit nondeterminism, resulting in the *depth-first search* (DFS) strategy for searching. Noted by Spivey [59], other search strategies can be implemented by other choices of functors. For example, *depth-bounded search* (DBS) can be implemented with the functor $\mathit{Int} \to [a]$, and *breadth-first search* (BFS) can be implemented with the functor $[[a]]$ (or Kidney and Wu [31]'s more efficient *LevelT* functor).

A powerful application of scoped effects is modelling search strategies:

$$\mathbf{data}\ \mathit{Strategy}\ x = \mathit{DFS}\ x \mid \mathit{BFS}\ x \mid \mathit{DBS}\ \mathit{Int}\ x$$

so that the programmer can freely specify the search strategy of nondeterministic choices in a scope. The algebraic signature *Choice* and scoped signature *Strategy* can be handled by a functorial algebra carried by the endofunctor $([a], [[a]], \mathit{Int} \to [a])$ and a base type $[a]$ (assuming that depth-first search is the default strategy). The complete code is in the supplementary material.

Example 6. A scoped operation for the effect of mutable state is the operation $\mathit{local}\ s\ p$ that executes the program p with a state s and restores to the original state after p finishes. Thus $(\mathit{local}\ s\ p \gg\!= k)$ is different from $\mathit{local}\ s\ (p \gg\!= k)$, and *local* should be modelled as a scoped operations of signature $\mathbf{data}\ \mathit{Local}\ s\ a = \mathit{Local}\ s\ a$. Together with the usual algebraic operations *get* and *put* of state, *Local* can be interpreted with a functorial algebra carried by the state monad $\mathbf{type}\ \mathit{State}\ s\ a = s \to (s, a)$. The essential part of the functorial algebra is the following *enterE* for *Local* (complete code in the supplementary material):

$$\begin{aligned}&\mathit{enterE} :: \mathit{Local}\ (\mathit{State}\ s\ (\mathit{State}\ s\ a)) \to \mathit{State}\ s\ a\\ &\mathit{enterE}\ (\mathit{Local}\ s'\ f)\ s = \mathbf{let}\ (\_, k) = f\ s\ \mathbf{in}\ k\ s\end{aligned}$$

Example 7. Parallel composition of processes is not an operation in the usual algebraic presentations of process calculi [61, 62] precisely because it not algebraic: $(p \mid q) \gg\!= k \neq (p \gg\!= k) \mid (q \gg\!= k)$. Again, we can model it as a scoped operation, and different scheduling behaviours of processes can be given as different functorial algebras. The supplementary material contains complete code of handling parallel composition using the so-called resumption monad [11, 47].

3 Categorical Foundations for Scoped Operations

We now move on to a categorical foundation for scoped effects and functorial algebras. First, we recall some standard category theory underlying algebraic effects and handlers (Section 3.1) and also Piróg et al. [46]'s monad P that models the syntax of scoped operations, which is exactly the *Prog* monad in the Haskell implementation (Section 3.2). Then, we define functorial algebras formally (Section 3.3) and show that there is an adjunction between the category of functorial algebras and the base category (Section 3.4) inducing the monad P, which provides a means to interpret the syntax of scoped operations.

The rest of this paper assumes familiarity with basic category theory, such as adjunctions, monads, and initial algebras, which are covered by standard texts [6, 41, 55]. The mathematical notation in this paper is summarised in the appendices, which may be consulted if the meaning of some symbols are unclear.

3.1 Syntax and Semantics of Algebraic Operations

The relationships between *equational theories*, *Lawvere theories*, *monads*, and *computational effects* are well-studied for decades from many perspectives [23, 30, 45, 48, 54, 57]. Here we recap a simplified version of equational theories by Kelly and Power [30] that we follow to model algebraic and scoped effects on *locally finitely presentable* (lfp) categories [1].

Locally Finitely Presentable Categories The use of lfp categories in this paper is limited to some standard results about the existence of many initial algebras in lfp categories, and thus a reader not familiar with lfp categories may follow this paper with some simple intuition: a category $\mathbb{C}$ is lfp if it has all (small) colimits and a set of *finitely presentable objects* such that every object in $\mathbb{C}$ can be obtained by 'glueing' (formally, as *filtered colimits* of) some finitely presentable objects. For example, `Set` is lfp with finite sets as its finitely presentable objects, and indeed every set can be obtained by glueing, here meaning taking the union of, all its finite subsets: $X = \bigcup \{ N \subseteq X \mid N \text{ finite} \}$. Other examples of lfp categories include the category of partially ordered sets, the category of graphs, the category of small categories, and presheaf categories (we refer the reader to the excellent exposition [57] for concrete examples), thus lfp categories are widespread to cover many semantic settings of programming languages.

Moreover, an endofunctor $F : \mathbb{C} \to \mathbb{C}$ is said to be *finitary* if it preserves 'glueing' (filtered colimits), which implies that its values FX are determined by its values at finitely presentable objects: $FX \cong F(\mathtt{colim}_i N_i) \cong \mathtt{colim}_i F N_i$ where N_i are the finitely presentable objects that generate X when glued together. For example, polynomial functors $\coprod_{n \in \mathbb{N}} Pn \times (-)^n$ on `Set` are finitary where Pn is a set for every n.

Algebraic Operations on LFP Categories Fixing an lfp category $\mathbb{C}$, we take finitary endofunctors $\Sigma : \mathbb{C} \to \mathbb{C}$ as signatures of operations on $\mathbb{C}$. Like in Section 2.1, the intuition is that every natural transformation $\coprod_{\mathbb{C}(R,-)} P \to \Sigma-$ for some object $P : \mathbb{C}$ and a finitely presentable object $R : \mathbb{C}$ stands for an operation taking a parameter of type P and R-many arguments. The category $\Sigma\text{-}\mathit{Alg}$ of Σ*-algebras* is defined as usual: it has pairs $\langle X : \mathbb{C}, \alpha : \Sigma X \to X \rangle$ as objects and morphisms $h : X \to X'$ such that $h \cdot \alpha = \alpha' \cdot \Sigma h$ as morphisms $\langle X, \alpha \rangle \to \langle X', \alpha' \rangle$. The following classical results (see e.g. [2, 5]) give sufficient conditions for constructing initial and free Σ-algebras:

Lemma 1. *If category $\mathbb{C}$ has finite coproducts and colimits of all ω-chains and functor $\Sigma : \mathbb{C} \to \mathbb{C}$ preserves them, then the forgetful functor $U_\Sigma : \Sigma\text{-}\mathit{Alg} \to \mathbb{C}$ forgetting the structure maps has a left adjoint $\mathit{Free}_\Sigma : \mathbb{C} \to \Sigma\text{-}\mathit{Alg}$ mapping every $X : \mathbb{C}$ to a Σ-algebra $\langle \Sigma^* X, op_X \rangle$ where $\Sigma^* X$ denotes the initial algebra $\mu Y.\, X + \Sigma Y$ and $op_X : \Sigma \Sigma^* X \to \Sigma^* X$.*

Lemma 1 is applicable to our setting since $\mathbb{C}$ being lfp directly implies that it has all colimits, and finitary functors Σ preserve colimits of ω-chains because colimits of ω-chains are filtered. Hence we have an adjunction: $\mathit{Free}_\Sigma \dashv U_\Sigma$:

$\Sigma\texttt{-}\mathtt{Alg} \to \mathbb{C}$. We denote the monad from the adjunction by $\Sigma^* = \mathtt{U}_\Sigma \mathtt{Free}_\Sigma$ (which is implemented as the *Free* Σ monad in Section 2.1). The idea is still that syntactic terms built from operations in Σ are modelled by the monad Σ^*, and semantics of operations are given by Σ-algebras. Given any Σ-algebra $\langle X, \alpha : \Sigma X \to X\rangle$ and morphism $g : A \to X$ in $\mathbb{C}$, they induce an interpretation morphism $\mathit{handle}_{\langle X,\alpha\rangle} g : \Sigma^* A \to X$ s.t.

$$\mathit{handle}_{\langle X,\alpha\rangle} g = \mathtt{U}_\Sigma(\epsilon_{\langle X,\alpha\rangle} \cdot \mathtt{Free}_\Sigma g) : \Sigma^* A = \mathtt{U}_\Sigma \mathtt{Free}_\Sigma A \to X \tag{9}$$

where $\epsilon_{\langle X,\alpha\rangle} : \mathtt{Free}_\Sigma \mathtt{U}_\Sigma \langle X, \alpha\rangle \to \langle X, \alpha\rangle$ is the counit of $\mathtt{Free}_\Sigma \dashv \mathtt{U}_\Sigma$.

Algebraic Effects and Handlers The perspective of Plotkin and Pretnar [52] is that computational effects are characterised by signatures Σ of primitive effectful operations, and they determine monads Σ^* that model programs syntactically. Then Σ-algebras are *handlers* [52] of operations that can be applied to programs using (9) to give specific semantics to operations.

The approach of algebraic effects has led to a significant body of research on programming with effects and handlers, but it imposes an assumption on the operations to be modelled: the construction of Σ^* in Lemma 1 [2, 5] implies that the multiplication μ of the monad Σ^* satisfies the *algebraicity* property: $\mathit{op} \cdot (\Sigma \circ \mu) = \mu \cdot (\mathit{op} \circ \Sigma^*) : \Sigma\Sigma^*\Sigma^* \to \Sigma^*$ where $\mathit{op} : \Sigma(\Sigma^*) \to \Sigma^*$. This intuitively means that every operation in Σ must be commutative with sequential composition of computations. Many, but not all, effectful operations satisfy this property, and they are called *algebraic operations.*

Adjoint Approach to Effects The crux of algebraic effects and handlers is the adjunction $\mathtt{Free}_\Sigma \dashv \mathtt{U}_\Sigma$. However, we have not relied on the adjunction being the free/forgetful one at all: given any monad $P : \mathbb{C} \to \mathbb{C}$ that models the syntax of effectful *P*rograms, if $L \dashv R : \mathbb{D} \to \mathbb{C}$ is an adjunction such that $RL \cong P$ as monads, then objects D in $\mathbb{D}$ provide a means to interpret programs PA—for any $g : A \to RD$ in $\mathbb{C}$, we have the following interpretation morphism

$$\mathit{handle}_D g = R(\epsilon_D \cdot Lg) : PA \cong R(LA) \to RD \tag{10}$$

The intuition for g is that it transforms the returned value A of a computation into the carrier RD, so it corresponds to the ‘value case’ of effect handlers [8]. Piróg et al. [46] call this approach the *adjoint-theoretic approach to syntax and semantics of effects*, and they construct an adjunction between *indexed algebras* and the base category for modelling scoped operations. Earlier, Levy [37] and Kammar and Plotkin [28] also adopt a similar adjunction-based viewpoint in the treatment of call-by-push-value calculi: *value types* are interpreted in the base category $\mathbb{C}$, and *computation types* are interpreted in the algebra category $\mathbb{D}$.

Remark 1. A notable missing part of our treatment is the *equations* that specify operations in a signature. Following Kelly and Power [30], an equation for a signature $\Sigma : \mathbb{C} \to \mathbb{C}$ can be formulated as a pair of monad morphisms $\sigma, \tau : \Gamma^* \to \Sigma^*$ for some finitary functor Γ, and taking their coequaliser $\Gamma^* \overset{\tau}{\underset{\sigma}{\rightrightarrows}} \Sigma^* \twoheadrightarrow M$ in the category of finitary monads constructs a monad M that represents terms modulo the equation $l = r$. Although it seems straightforward to extend this formulation of equational theories work with scoped effects, we do not consider equations in this paper for the sake of simplicity.

Remark 2. Working with lfp categories precludes operations with infinite arguments, such as the *get* operation (1) of mutable state when the state has infinite possible values, but this limitation is not inherent and can be handled by moving to *locally κ-presentable categories* [1] for some larger cardinal κ.

3.2 Syntax of Scoped Operations

Not all operations in programming languages can be adequately modelled as algebraic operations on `Set`, for example, λ-abstraction [16], memory cell generation [38, 48], more generally, effects with dynamically generated instances [62], explicit substitution [18], channel restriction in π-calculus [61], and their syntax are usually modelled in some functor categories. More recently, Piróg et al. [46] extend Ghani and Uustalu [18]'s work to model a family of non-algebraic operations, which they call *scoped operations.* In this subsection, we review their development in the setting of lfp categories. Throughout the rest of the paper, we fix an lfp category $\mathbb{C}$, and refer to it as the *base category*, and it is intended to be the category in which types of a programming language are interpreted. Furthermore, we fix two finitary endofunctors $\Sigma, \Gamma : \mathbb{C} \to \mathbb{C}$ and call them the *algebraic signature* and *scoped signature* respectively.

Syntax Endofunctor P Now our goal is to construct a monad $P : \mathbb{C} \to \mathbb{C}$ that models the syntax of programs with algebraic operations in Σ and non-algebraic scoped operations in Γ. First we construct its underlying endofunctor. When $\mathbb{C}$ is `Set`, the intuition for programs PA is that they are terms inductively built from the following inference rules:

$$\frac{a \in A}{var(a) \in PA} \qquad \frac{o \in \Sigma n \quad k : n \to PA}{o(k) \in PA} \qquad \frac{s \in \Gamma n \quad p : n \to PX \quad k : X \to PA}{\{s(p); k\} \in PA}$$

where n ranges over finite sets and $o \in \Sigma n$ represents an algebraic operation of $|n|$ arguments, and similarly $s \in \Gamma n$ is a scoped operation that creates $|n|$ scopes. The difference between algebraic and scoped operations is manifested by an additional explicit continuation k in the third rule, as it is *not* the case that sequentially composing $s(p)$ with k equals $s(p; k)$ like for algebraic operations, so the continuation for scoped operations must be explicitly kept in the syntax. When $\mathbb{C}$ is any lfp category, these rules translate to the following recursive equation for the functor $P : \mathbb{C} \to \mathbb{C}$:

$$PA \cong A + \Sigma(PA) + \int^{X:\mathbb{C}} \coprod_{\mathbb{C}(X,PA)} \Gamma(PX) \tag{11}$$

where the existentially quantified X in the third rule is translated to a *coend* $\int^{X:\mathbb{C}}$ in $\mathbb{C}$ [41]. Moreover, the coend in (11) is isomorphic to $\Gamma(P(PA))$ because by the coend formula of Kan extension, it exactly computes $\mathtt{Lan}_I(\Gamma P)(PA)$, i.e. the left Kan-extension of ΓP along the identity functor $I : \mathbb{C} \to \mathbb{C}$, and by definition $\mathtt{Lan}_I(\Gamma P) = \Gamma P$. Thus (11) is equivalent to

$$PA \cong A + \Sigma(PA) + \Gamma(P(PA)) \tag{12}$$

which is exactly the *Prog Σ Γ* datatype that we saw in the Haskell implementation (7). To obtain a solution to (12), we construct a (higher-order) endofunctor $G : \mathit{Endo}_f(\mathbb{C}) \to \mathit{Endo}_f(\mathbb{C})$ to represent the *G*rammar where $\mathit{Endo}_f(\mathbb{C})$ is the category of finitary endofunctors on $\mathbb{C}$:

$$G = \mathit{Id} + \Sigma \circ - + \Gamma \circ - \circ - \tag{13}$$

where $\mathit{Id} : \mathbb{C} \to \mathbb{C}$ is the identity functor. Then Lemma 1 is applicable because $\mathit{Endo}_f(\mathbb{C})$ has all small colimits since colimits in functor categories can be computed pointwise and $\mathbb{C}$ has all small colimits. Furthermore, G preserves all filtered colimits, in particular colimits of ω-chains, because $- \circ = : \mathit{Endo}_f(\mathbb{C}) \times \mathit{Endo}_f(\mathbb{C}) \to \mathit{Endo}_f(\mathbb{C})$ is finitary following from direct verification. Since initial algebras are precisely free algebras generated by the initial object, by Lemma 1, there is an initial G-algebra $\langle P : \mathit{Endo}_f(\mathbb{C}), \mathit{in} : GP \to P\rangle$ and in is an isomorphism. Thus P obtained in this way is indeed a solution to (12)—the endofunctor modelling the syntax of programs with algebraic and scoped operations.

Monadic Structure of P Next we equip the endofunctor P with a monad structure. This can be done in several ways, either by the general result about *Σ-monoids* [14, 16] in $\mathit{Endo}_f(\mathbb{C})$, or by [43, Theorem 4.3], or by the following relatively straightforward argument in [46]: by the 'diagonal rule' of computing initial algebras by Backhouse et al. [4], $P = \mu G$ (13) is isomorphic to $P' = \mu X.\ \mathit{Id} + \Sigma \circ X + \Gamma \circ P \circ X$. Note that P' is exactly $(\Sigma + \Gamma \circ P)^*$ as endofunctors by Lemma 1, thus

$$P \cong (\Sigma + \Gamma \circ P)^* : \mathit{Endo}_f(\mathbb{C}) \tag{14}$$

Then we equip P with the same monad structure as the ordinary free monad $(\Sigma + \Gamma \circ P)^*$. The implementation in (7) is exactly this monad structure.

3.3 Functorial Algebras of Scoped Operations

To interpret the monad P (12) modelling the syntax of scoped operations, it is natural to expect that semantics is given by G-algebras on $\mathit{Endo}_f(\mathbb{C})$ so that interpretation is then the catamorphisms from μG to G-algebras. And following the adjoint-theoretic approach (10), we would like to have an adjunction $\begin{tikzcd} G\text{-}\mathit{Alg} \arrow[r, phantom, "\perp"] & \mathbb{C} \end{tikzcd}$ such that the induced monad is isomorphic to P. However, there seems no natural way to construct such an adjunction unless we replace G-algebras with a slight extension of it, which we referred to as *functorial algebras*, as the notion for giving semantics to scoped operations. In the following, we first define functorial algebras formally (Definition 1) and then show the adjunction between the category of functorial algebras and the base category (Theorem 1), which allows us to interpret P with functorial algebras.

A functorial algebra is carried by an endofunctor $H : \mathbb{C} \to \mathbb{C}$ with additionally an object X in $\mathbb{C}$. The endofunctor H also comes with a morphism $\alpha^G : GH \to H$ in $\mathit{Endo}_f(\mathbb{C})$, and the object X is equipped with a morphism $\alpha^I : \Sigma X + \Gamma HX \to X$ in $\mathbb{C}$. The intuition is that given a program of type $PX \cong X + \Sigma(PX) + \Gamma(P(PX))$, the middle P in ΓPP corresponds to the part of a program enclosed by some scoped operations (i.e. the p in $\{s(p) \gg\!= k\}$), and this part of the program is interpreted by H with α^G. After the enclosed part is interpreted, α^I interprets the outermost layer of the program by X with α^I in the same way as interpreting free monads of algebraic operations. More precisely, let $I : \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathbb{C}$ be a bi-functor such that [3]

$$I_H X = \Sigma X + \Gamma(HX) \qquad\qquad I_\sigma f = \Sigma f + \Gamma(\sigma \cdot Hf) \tag{15}$$

for all $H : \mathit{Endo}_f(\mathbb{C})$ and $X : \mathbb{C}$ and all morphisms $\sigma : H \to H'$ and $f : X \to X'$. Then we define an endofunctor $\mathit{Fn} : \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$ such that

$$\mathit{Fn}\langle H, X\rangle = \langle GH, I_H X\rangle \tag{16}$$

Definition 1. *A* functorial algebra *is an object $\langle H, X\rangle$ in $\mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$ paired with a structure map $\mathit{Fn}\langle H, X\rangle \to \langle H, X\rangle$, or equivalently it is a quadruple*

$$\big\langle H : \mathit{Endo}_f(\mathbb{C}), \quad X : \mathbb{C}, \quad \alpha^G : GH \to H, \quad \alpha^I : \Sigma X + \Gamma(HX) \to X \big\rangle$$

where $GH = \mathit{Id} + \Sigma \circ H + \Gamma \circ H \circ H$. Morphisms between two functorial algebras $\langle H_1, X_1, \alpha_1^G, \alpha_1^I\rangle$ and $\langle H_2, X_2, \alpha_2^G, \alpha_2^I\rangle$ are pairs $\langle \sigma : H_1 \to H_2,\ f : X_1 \to X_2\rangle$ making the following diagrams commute:

$$\begin{tikzcd} GH_1 \arrow[r, "\alpha_1^G"] \arrow[d, "G\sigma"'] & H_1 \arrow[d, "\sigma"] \\ GH_2 \arrow[r, "\alpha_2^G"'] & H_2 \end{tikzcd} \qquad \begin{tikzcd} \Sigma X_1 + \Gamma(H_1 X_1) \arrow[r, "\alpha_1^I"] \arrow[d, "\Sigma f + \Gamma(\sigma \circ f)"'] & X_1 \arrow[d, "f"] \\ \Sigma X_2 + \Gamma(H_2 X_2) \arrow[r, "\alpha_2^I"'] & X_2 \end{tikzcd}$$

Functorial algebras and their morphisms form a category **Fn-Alg**.

Example 8. We reformulate our programming example of nondeterministic choice with *once* shown Example 4 in the formal definition. Let $\mathbb{C} = \mathtt{Set}$ in this example and $1 = \{\star\}$ be some singleton set. We define signature endofunctors

$$\Sigma X = 1 + X \times X \qquad\qquad \Gamma X = X$$

so that Σ represents nullary algebraic operation *fail* and binary algebraic operation *or*, and Γ represents the unary scoped operation *once* that creates one scope. Let *List* : $\mathtt{Set} \to \mathtt{Set}$ be the endofunctor mapping a set X to the set of finite lists

[3] The first argument H to I is written as subscript so that we have a more compact notation I_H^* when taking the free monad of $I_H : \mathbb{C}^{\mathbb{C}}$ with the first argument fixed.

with elements from X. We define natural transformations $\alpha^{\Sigma} : \Sigma \circ \mathit{List} \to \mathit{List}$ and $\alpha^{\Gamma} : \Gamma \circ \mathit{List} \circ \mathit{List} \to \mathit{List}$ by

$$\alpha^{\Sigma}_X(\iota_1\ \star) = \mathit{nil}, \quad \alpha^{\Sigma}_X(\iota_2\ \langle x, y\rangle) = x +\!\!+ y, \quad \alpha^{\Gamma}_X(\mathit{nil}) = \mathit{nil}, \quad \alpha^{\Gamma}_X(\mathit{cons}\ x\ \mathit{xs}) = x$$

where nil is the empty list; $+\!\!+$ is list concatenation; and $\mathit{cons}\ x\ \mathit{xs}$ is the list with an element x in front of xs. Then for any set X, $\langle \mathit{List}, \mathit{List}\ X\rangle$ carries a functorial algebra with structure maps

$$\alpha^G = [\eta^{\mathit{List}}, \alpha^{\Sigma}, \alpha^{\Gamma}] : G\mathit{List} \to \mathit{List} \qquad \alpha^I = [\alpha^{\Sigma}_X, \alpha^{\Gamma}_X] : I_{\mathit{List}}X \to X \qquad (17)$$

where $\eta^{\mathit{List}} : \mathit{Id} \to \mathit{List}$ wraps any element into a singleton list.

The last example exhibits that one can define a functorial algebra carried by $\langle H, HX\rangle$ from a G-algebra on $H : \mathit{Endo}_f(\mathbb{C})$ by simply choosing the object component to be HX for an arbitrary $X : \mathbb{C}$. In other words, there is a faithful functor $G\text{-}\mathit{Alg} \to \mathit{Fn}\text{-}\mathit{Alg}$, which results in functorial algebras that interpret the outermost layer of a program—the part not enclosed by any scoped operation—in the same way as the inner layers. But in general, the object component of functorial algebras offers the flexibility that the outermost layer can be interpreted differently from the inner layers, as in the following example.

Example 9. Continuing Example 8, if one is only interested in the final number of possible outcomes, then one can define a functorial algebra $\langle \mathit{List}, \mathbb{N}, \alpha^G, \alpha^I\rangle$ where α^G is (17) and $\alpha^I(\iota_1\ (\iota_1\star)) = 0$,

$$\alpha^I(\iota_1\ (\iota_2\langle x, y\rangle)) = x + y, \quad \alpha^I(\iota_2\ \mathit{nil}) = 0, \quad \alpha^I(\iota_2\ (\mathit{cons}\ n\ \mathit{ns})) = n$$

3.4 Interpreting with Functorial Algebras

In the rest of this section we show how functorial algebras can be used to interpret programs PA (12) with scoped operations. We first construct a simple adjunction $\uparrow \dashv \downarrow$ between the base category $\mathbb{C}$ and $\mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$, which is then composed with the free/forgetful adjunction $\mathit{Free}_{\mathit{Fn}} \dashv U_{\mathit{Fn}}$ between $\mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$ and $\mathit{Fn}\text{-}\mathit{Alg}$ for the functor Fn (16). The resulting adjunction (18) is proven to induce a monad T isomorphic to P (Theorem 1), and by the adjoint-theoretic approach to syntax and semantics (10), this adjunction provides a means to interpret scoped operations modelled with the monad P (Theorem 2).

First we define functor $\uparrow : \mathbb{C} \to \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$ such that $\uparrow X = \langle 0, X\rangle$ where $0 : \mathit{Endo}_f(\mathbb{C})$ is the initial endofunctor—the constant functor sending everything to the initial object in $\mathbb{C}$. The functor $\uparrow$ is left adjoint to the projection functor $\downarrow : \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathbb{C}$ of the second component.

Then we would like to compose $\uparrow \dashv \downarrow$ with the free-forgetful adjunction $\mathit{Free}_{\mathit{Fn}} \dashv U_{\mathit{Fn}}$ for the endofunctor Fn (16) on $\mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$, and the latter adjunction indeed exists.

Lemma 2. *The endofunctor Fn (16) on $\mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$ has free algebras, i.e. there is a functor $\mathit{Free}_{\mathit{Fn}} : \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathit{Fn}\text{-}\mathit{Alg}$ left adjoint to the forgetful functor $U_{\mathit{Fn}} : \mathit{Fn}\text{-}\mathit{Alg} \to \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C}$.*

These two adjunctions are depicted in the following diagram:

$$\begin{tikzcd}
\mathtt{Fn\text{-}Alg} \arrow[r, shift right, "U_{\mathtt{Fn}}"'] \arrow[r, phantom, "\perp"] & \mathtt{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, shift right, "\mathtt{Free}_{\mathtt{Fn}}"'] \arrow[r, shift right, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right, "\uparrow"'] \arrow[loop right, "T"]
\end{tikzcd} \tag{18}$$

and we compose them to obtain an adjunction $\mathtt{Free}_{\mathtt{Fn}} \uparrow \dashv \downarrow U_{\mathtt{Fn}}$ between $\mathtt{Fn\text{-}Alg}$ and $\mathbb{C}$, giving rise to a monad $T = \downarrow U_{\mathtt{Fn}} \mathtt{Free}_{\mathtt{Fn}} \uparrow$. In the rest of this section, we prove that T is isomorphic to P (11) in the category of monads, which is crucial in this paper, since it allows us to interpret scoped operations modelled by the monad P with functorial algebras $\mathtt{Fn\text{-}Alg}$.

We first establish a technical lemma characterising the free $\mathtt{Fn}$-algebra on the product category $\mathtt{Endo}_f(\mathbb{C}) \times \mathbb{C}$ in terms of the free algebras in $\mathbb{C}$ and $\mathtt{Endo}_f(\mathbb{C})$.

Lemma 3. *There is a natural isomorphism between $\mathtt{Free}_{\mathtt{Fn}}$ and the following*

$$\widehat{\mathtt{Free}}_{\mathtt{Fn}}\langle H, X\rangle = \left\langle G^*H : \mathtt{Endo}_f(\mathbb{C}), \quad (I_{G^*H})^*X : \mathbb{C}, \quad op_H^{G^*}, \quad op_X^{(I_{G^*H})^*} \right\rangle$$

where $op_H^{G^*} : G(G^*H) \to G^*H$ and $op_X^{(I_{G^*H})^*} : I_{G^*H}((I_{G^*H})^*X) \to (I_{G^*H})^*X$ are the structure maps of the free G-algebra and I_{G^*H}-algebra respectively.

Theorem 1. *Monads P (12) and T (18) are isomorphic as monads.*

Remark 3. In general, the right adjoint $\downarrow U_{\mathtt{Fn}}$ is *not* monadic since it does not reflect isomorphisms, which is a necessary condition for it to be monadic by Beck's monadicity theorem [41]. This entails that the category $\mathtt{Fn\text{-}Alg}$ of functorial algebras is not equivalent to the category of Eilenberg-Moore algebras. Nonetheless, as we will see later in Section 4, functorial algebras and Eilenberg-Moore algebras have the same expressive power for interpreting scoped operations.

The isomorphism established Theorem 1 enables us to interpret programs modelled by the monad P using functorial algebras following (10): for any functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$ (Definition 1), and any morphism $g : A \to X$ in the base category $\mathbb{C}$, there is a morphism

$$handle_{\langle H,X,\alpha^G,\alpha^I\rangle}\ g = \downarrow U_{\mathtt{Fn}}(\epsilon_{\langle H,X,\alpha^G,\alpha^I\rangle} \cdot \mathtt{Free}_{\mathtt{Fn}} \uparrow g) : TA \cong PA \to X \tag{19}$$

which interprets programs PA with the functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$. Furthermore, we can derive the following recursive formula (20) for this interpretation morphism, which is exactly the Haskell implementation in Figure 1.

Theorem 2 (Interpreting with Functorial Algebras). *For any functorial algebra $\alpha = \langle H, X, \alpha^G, \alpha^I\rangle$ as in Definition 1, and any morphism $g : A \to X$ for some A in the base category $\mathbb{C}$, let $h = (\!|\alpha^G|\!) : P \to H$ be the catamorphism from the initial G-algebra P to the G-algebra $\alpha^G : GH \to H$. The interpretation of PA with this algebra α and g satisfies*

$$handle_\alpha\ g = [g,\quad \alpha^I_\Sigma \cdot \Sigma(handle_\alpha\ g),\quad \alpha^I_\Gamma \cdot \Gamma h_X \cdot \Gamma P(handle_\alpha\ g)] \cdot in^\circ_A \tag{20}$$

where $in^\circ : P \to \mathtt{Id} + \Sigma \circ P + \Gamma \circ P \circ P$ is the isomorphism between P and GP; morphisms $\alpha^I_\Sigma = \alpha^I \cdot \iota_1 : \Sigma X \to X$ and $\alpha^I_\Gamma = \alpha^I \cdot \iota_2 : \Gamma H X \to X$ are the two components of $\alpha^I : \Sigma X + \Gamma H X \to X$.

To summarise, we have defined a notion of functorial algebras that we use to handle scoped operations. The heart of the development is the adjunction (18) that induces a monad isomorphic to the monad P (12) that models the syntax of programs with scoped operations, following which we derive a recursive formula (20) that interprets programs with functor algebras. The formula is exactly the implementation in Figure 1: the datatype *EndoAlg* represents the α^G in (20); datatype *BaseAlg* corresponds to α^I; function *hcata* implements $(\!|\alpha^G|\!)$.

4 Comparing the Models of Scoped Operations

Functorial algebras are not the only option for interpreting scoped operations. In this section we compare functorial algebras with two other approaches, one being Piróg et al. [46]'s *indexed algebras* and the other one being *Eilenberg-Moore* (EM) algebras of the monad P (12), which simulate scoped operations with algebraic operations. After a brief description of these two kinds of algebras, we compare them and show that their expressive power is in fact equivalent.

4.1 Interpreting Scoped Operations with Eilenberg-Moore Algebras

In standard algebraic effects, handlers are just Σ-algebras for some signature functor $\Sigma : \mathbb{C} \to \mathbb{C}$, and it is well known that the category $\Sigma\text{-}\mathit{Alg}$ of Σ-algebras is equivalent to the category $\mathbb{C}^{\Sigma^*}$ of EM algebras of the monad Σ^*. Thus handlers of algebraic operations are exactly EM algebras of the monad Σ^* modelling the syntax of algebraic operations. This observation suggests that we may also use EM algebras of the monad P (12) as the notion of handlers for scoped operations.

Lemma 4. *EM algebras of P are equivalent to $(\Sigma + \Gamma \circ P)$-algebras. In other words, an EM algebra of P is equivalently a tuple*

$$\langle X : \mathbb{C},\ \alpha_\Sigma : \Sigma X \to X,\ \alpha_\Gamma : \Gamma(PX) \to X\rangle \tag{21}$$

Thus we obtain a way of interpreting scoped operations based on the adjunction $\mathit{Free}_{\Sigma+\Gamma\circ P} \dashv U_{\Sigma+\Gamma\circ P}$: given an EM algebra $\alpha = \langle X, \alpha_\Sigma, \alpha_\Gamma\rangle$ of P as in (21), then for any $A : \mathbb{C}$ and morphism $g : A \to X$, the interpretation of PA by g and this EM algebra is

$$\mathit{handle}_\alpha\ g = U_{\Sigma+\Gamma\circ P}(\epsilon_\alpha \cdot \mathit{Free}_{\Sigma+\Gamma\circ P}\ g) : PA \cong (\Sigma + \Gamma \circ P)^* A \to X \tag{22}$$

The formula (22) can also be turned into a recursive form:

$$\mathit{handle}_\alpha\ g = [g,\ \ \alpha_\Sigma \cdot \Sigma(\mathit{handle}_\alpha\ g),\ \ \alpha_\Gamma \cdot \Gamma P(\mathit{handle}_\alpha\ g)] \cdot \mathit{in}^\circ_A \tag{23}$$

that suits implementation (see the appendices for more details).

Interpreting scoped operation with EM algebras can be understood as simulating scoped operations with algebraic operations and general recursion: a signature (Σ, Γ) of algebraic-and-scoped operations is simulated by a signature $(\Sigma+\Gamma\circ P)$ of algebraic operations where P is recursively given by $(\Sigma+\Gamma\circ P)^*$. In this way, one can simulate scoped operation in languages implementing algebraic effects that allow signatures of operation to be recursive, such as [7, 19, 36], but not the original design by Plotkin and Pretnar [52], which requires signatures of operations to mention only some *base types*.

The downside of this simulating approach is that the denotational semantics of the language becomes more complex and usually involves solving some domain-theoretic recursive equations, like in [7]. Moreover, this approach typically requires handlers to be defined with general recursion, which obscures the inherent structure of scoped operations, making reasoning about handlers of scoped operations more difficult.

4.2 Indexed Algebras of Scoped Effects

Indexed algebras of scoped operations by Piróg et al. [46] are yet another way of interpreting scoped operations. They are based on the following adjunction:

$$\begin{tikzcd} \mathtt{Ix\text{-}Alg} \arrow[r, bend right, "U_{Ix}"'] \arrow[r, phantom, "\perp"] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, bend right, "Free_{Ix}"'] \arrow[r, bend right, "\downharpoonleft"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\upharpoonleft"'] \end{tikzcd} \tag{24}$$

where $\mathbb{C}^{|\mathbb{N}|}$ is the functor category from the discrete category $|\mathbb{N}|$ of natural numbers to the base category $\mathbb{C}$. That is to say, an object in $\mathbb{C}^{|\mathbb{N}|}$ is a family of objects A_i in $\mathbb{C}$ indexed by natural numbers $i \in |\mathbb{N}|$, and a morphism $\tau : A \to B$ in $\mathbb{C}^{|\mathbb{N}|}$ is a family of morphisms $\tau_i : A_i \to B_i$ in $\mathbb{C}$ (with no coherence conditions). An endofunctor $\mathtt{Ix} : \mathbb{C}^{|\mathbb{N}|} \to \mathbb{C}^{|\mathbb{N}|}$ is defined to characterise indexed algebras:

$$\mathtt{Ix}A = \Sigma \circ A + \Gamma \circ (\triangleleft A) + (\triangleright A)$$

where $\triangleleft$ and $\triangleright$ are functors $\mathbb{C}^{|\mathbb{N}|} \to \mathbb{C}^{|\mathbb{N}|}$ *shifting indices* such that $(\triangleleft A)_i = A_{i+1}$ and $(\triangleright A)_0 = 0$ and $(\triangleright A)_{i+1} = A_i$. Then objects in $\mathtt{Ix\text{-}Alg}$ are called *indexed algebras*. Furthermore, since a morphism $(\triangleright A) \to A$ is in bijection with $A \to (\triangleleft A)$, an indexed algebra can be given by the following tuple:

$$\langle A : \mathbb{C}^{|\mathbb{N}|}, \quad a : \Sigma \circ A \to A, \quad d : \Gamma(\triangleleft A) \to A, \quad p : A \to \triangleleft A \rangle \tag{25}$$

The operational intuition for it is that the carrier A_i at level i interprets the part of syntax enclosed by i layers of scopes, and when interpreting a scoped operation $\Gamma(P(PX))$ at layer i, the part of syntax outside the scope is first interpreted, resulting in $\Gamma(PA_i)$, and then the indexed algebra provides a way p to $p$romote the carrier to the next level, resulting in $\Gamma(PA_{i+1})$. After the inner layer is also interpreted as ΓA_{i+1}, the indexed algebra provides a way d to $d$emote the carrier, producing A_i again. Additionally the morphism a interprets ordinary $a$lgebraic operations.

Example 10. Example 8 for nondeterministic choice with *once* can be expressed with an indexed algebra as follows. For any set X, we define an indexed object $A : \mathbb{C}^{|\mathbb{N}|}$ by $A_0 = \mathit{List}\ X$ and $A_{i+1} = \mathit{List}\ A_i$. The object A carries an indexed algebra with the following structure maps: for all $i \in \mathbb{N}$, $a_i(\iota_1\ \star) = \mathit{nil}$ and

$$a_i(\iota_2\ \langle x, y\rangle) = x +\!\!+ y, \quad d_i(\mathit{nil}) = \mathit{nil}, \quad d_i(\mathit{cons}\ x\ xs) = x, \quad p_i(x) = \mathit{cons}\ x\ \mathit{nil}$$

The adjunction $\mathit{Free}_{\mathit{Ix}} \dashv U_{\mathit{Ix}}$ in (24) is the free-forgetful adjunction for Ix on $\mathbb{C}^{|\mathbb{N}|}$. The other adjunction $\upharpoonleft \dashv \downharpoonright$ is given by $\downharpoonright A = A_0$, $(\upharpoonleft X)_0 = X$, and $(\upharpoonleft X)_{i+1} = 0$ for all $i \in \mathbb{N}$. Importantly, Piróg et al. [46] show that the monad induced by the adjunction (24) is isomorphic to monad P (12), thus indexed algebras can also be used to interpret scoped operations

$$\mathit{handle}_{\langle A,a,d,p\rangle}\ g = \downharpoonright U_{\mathit{Ix}}(\epsilon_{\langle A,a,d,p\rangle} \cdot \mathit{Free}_{\mathit{Ix}} \upharpoonleft g) \tag{26}$$

in the same way as what we do for functorial algebras in Section 3.4. Interpreting with indexed algebras can also be implemented in Haskell with GHC's `DataKinds` extension for type-level natural numbers (which can be found in the appendices).

4.3 Comparison of Resolutions

Now we come back to the real subject of this section—comparing the expressivity of the three ways for interpreting scoped operations. Specifically, we construct *comparison functors* between the respective categories of the three kinds of algebras, which translate one kind of algebras to another in a way *preserving the induced interpretation* in the base category. Categorically, the three kinds of algebras correspond to three *resolutions* of the monad P, which form a category of resolutions (Definition 2) with comparison functors as morphisms. In this category, the Eilenberg-Moore resolution is the terminal object, and thus it automatically gives us comparison functors translating other kinds of algebras to EM algebras. To complete the circle of translations, we then construct comparison functors $K^{\mathrm{EM}}_{\mathrm{Fn}} : \mathbb{C}^P \to \mathit{Fn\text{-}Alg}$ translating EM algebras to functorial ones (Section 4.4) and $K^{\mathrm{Fn}}_{\mathrm{Ix}} : \mathit{Fn\text{-}Alg} \to \mathit{Ix\text{-}Alg}$ translating functorial algebras to indexed ones (Section 4.5).

Definition 2 (Resolutions [35]). *Given a monad M on $\mathbb{C}$, the category $\mathit{Res}(M)$ of* resolutions *of M has as objects adjunctions $\langle \mathbb{D}, L \dashv R : \mathbb{D} \to \mathbb{C}, \eta, \epsilon\rangle$ whose induced monad RL is M. A morphism from a resolution $\langle \mathbb{D}, L \dashv R, \eta, \epsilon\rangle$ to $\langle \mathbb{D}', L' \dashv R', \eta', \epsilon'\rangle$ is a functor $K : \mathbb{D} \to \mathbb{D}'$, called a* comparison functor*, such that it commutes with the left and right adjoints, i.e. $KL = L'$ and $R'K = R$.*

We have seen adjunctions for indexed algebras, EM algebras and functorial algebras respectively, each inducing the monad P up to isomorphism, so each of them can be identified with an object in the category $\mathit{Res}(E)$. For each resolution $\langle \mathbb{D}, L, R, \eta, \epsilon\rangle$, we have been using the objects D in $\mathbb{D}$ to interpret scoped operations modelled by P: for any morphism $g : A \to RD$ in $\mathbb{C}$, the interpretation of PA by D and g is $\mathit{handle}_D\ g = R(\epsilon_D \cdot Lg) : PA = RLA \to RD$. Crucially, we show that interpretations are preserved by comparison functors.

Lemma 5 (Preservation of Interpretation). *Let $K : \mathbb{D} \to \mathbb{D}'$ be any comparison functor between resolutions $\langle \mathbb{D}, L, R, \eta, \epsilon\rangle$ and $\langle \mathbb{D}', L', R', \eta', \epsilon'\rangle$ of some monad $M : \mathbb{C} \to \mathbb{C}$. For any object D in $\mathbb{D}$ and any $g : A \to RD$ in $\mathbb{C}$,*

$$\mathit{handle}_D\ g = \mathit{handle}_{KD}\ g : MA \to RD (= R'KD) \tag{27}$$

where each side interprets MA using $L \dashv R$ and $L' \dashv R'$ respectively.

This lemma implies that if there is a comparison functor K from some resolution $L \dashv R : \mathbb{D} \to \mathbb{C}$ to $L' \dashv R' : \mathbb{D}' \to \mathbb{C}$ of the monad P, then K can *translate* a $\mathbb{D}$ object to a $\mathbb{D}'$ object that preserves the induced interpretation. Thus the expressive power of $\mathbb{D}$ for interpreting P is not greater than $\mathbb{D}'$, in the sense that every $\mathit{handle}_D\ g$ that one can obtain from D in $\mathbb{D}$ can also be obtained by an algebra KD in $\mathbb{D}'$. Thus the three kinds of algebras for interpreting scoped operations have the same expressivity if we can construct a circle of comparison functors between their categories, which is what we do in the following.

Translating to EM Algebras As shown in [41], an important property of the Eilenberg-Moore adjunction is that it is the terminal object in the category $\mathtt{Res}(M)$ for any monad M, which means that there *uniquely exists* a comparison functor from *every* resolution to the Eilenberg-Moore resolution. Specifically, given a resolution $\langle \mathbb{D}, L, R, \eta, \epsilon \rangle$ of a monad M, the unique comparison functor K from $\mathbb{D}$ to the category $\mathbb{C}^M$ of the Eilenberg-Moore algebras is

$$KD = \big(M(RD) = RLRD \xrightarrow{R\epsilon_D} RD\big) \qquad \text{and} \qquad K(D \xrightarrow{f} D') = Rf$$

Lemma 6. *There uniquely exist comparison functors $K^{Ix}_{EM} : \mathtt{Ix\text{-}Alg} \to \mathbb{C}^P$ and $K^{Fn}_{EM} : \mathtt{Fn\text{-}Alg} \to \mathbb{C}^P$ from the resolutions of indexed algebras and functorial algebras to the resolution of EM algebras.*

4.4 Translating EM Algebras to Functorial Algebras

Now we construct a comparison functor $K^{\mathtt{EM}}_{\mathtt{Fn}} : \mathbb{C}^P \to \mathtt{Fn\text{-}Alg}$ translating EM algebras to functorial ones. The idea is straightforward: given an EM algebra X, we map it to the functorial algebra with X for interpreting the outermost layer and the functor P for interpreting the inner layers, which essentially leaves the inner layers uninterpreted before they get to the outermost layer.

Since $\mathbb{C}^P$ is isomorphic to $(\Sigma + \Gamma \circ P)\mathtt{\text{-}Alg}$, we can define $K^{\mathtt{EM}}_{\mathtt{Fn}}$ on $(\Sigma + \Gamma \circ P)$-algebras instead. Given any $\langle X : \mathbb{C}, \alpha : (\Sigma + \Gamma \circ P)X \to X\rangle$, it is mapped by $K^{\mathtt{EM}}_{\mathtt{Fn}}$ to the functorial algebra

$$\langle P,\ X,\ \mathit{in} : GP \to P,\ \alpha : (\Sigma + \Gamma \circ P)X \to X\rangle$$

and for any morphism f in $(\Sigma + \Gamma \circ P)\mathtt{\text{-}Alg}$, it is mapped to $\langle id_P, f\rangle$. To show $K^{\mathtt{EM}}_{\mathtt{Fn}}$ is a comparison functor, we only need to show that it commutes with the left and right adjoints of both resolutions. Details can be found in the appendices.

Lemma 7. *Functor K^{EM}_{Fn} is a comparison functor from the Eilenberg-Moore resolution of P to the resolution $\mathtt{Free}_{Fn}{\uparrow} \dashv {\downarrow}\mathtt{U}_{Fn}$ of functorial algebras.*

4.5 Translating Functorial Algebras to Indexed Algebras

At this point we have comparison functors $\mathtt{Ix\text{-}Alg} \xrightarrow{K^{\mathtt{Ix}}_{\mathtt{EM}}} \mathbb{C}^P \xrightarrow{K^{\mathtt{EM}}_{\mathtt{Fn}}} \mathtt{Fn\text{-}Alg}$. To complete the circle of translations, we construct a comparison functor $K^{\mathtt{Fn}}_{\mathtt{Ix}}$:

$\mathit{Fn\text{-}Alg} \to \mathit{Ix\text{-}Alg}$ in this subsection. The idea of this translation is that given a functorial algebra carried by endofunctor $H : \mathbb{C}^{\mathbb{C}}$ and object $X : \mathbb{C}$, we map it to an indexed algebra by iterating the endofunctor H on X. More precisely, $K^{\mathrm{Fn}}_{\mathrm{Ix}} : \mathit{Fn\text{-}Alg} \to \mathit{Ix\text{-}Alg}$ maps a functorial algebra

$$\langle H : \mathbb{C}^{\mathbb{C}},\ X : \mathbb{C},\ \alpha^G : \mathit{Id} + \Sigma \circ H + \Gamma \circ H \circ H \to H,\ \alpha^I : \Sigma X + \Gamma H X \to X \rangle$$

to an indexed algebra carried by $A : \mathbb{C}^{|\mathbb{N}|}$ such that $A_i = H^i X$, i.e. iterating H i-times on X. The structure maps of this indexed algebra $\langle a : \Sigma A \to A,\ d : \Gamma(\triangleleft A) \to A,\ p : A \to (\triangleleft A)\rangle$ are given by

$$\begin{aligned} a_0 &= (\alpha^I \cdot \iota_1) : \Sigma X \to X & a_{i+1} &= (\alpha^G_{H^i X} \cdot \iota_2) : \Sigma H H^i X \to H^{i+1} X \\ d_0 &= (\alpha^I \cdot \iota_2) : \Gamma H X \to X & d_{i+1} &= (\alpha^G_{H^i X} \cdot \iota_3) : \Gamma H H H^i X \to H^{i+1} X \end{aligned}$$

and $p_i = \alpha^G_{H^i X} \cdot \iota_1 : H^i X \to H H^i X$. On morphisms, $K^{\mathrm{Fn}}_{\mathrm{Ix}}$ maps a morphism $\langle \tau : H \to H', f : X \to X' \rangle$ in $\mathit{Fn\text{-}Alg}$ to $\sigma : H^i X \to H'^i X'$ in $\mathit{Ix\text{-}Alg}$ such that $\sigma_0 = f$ and $\sigma_{i+1} = \tau \circ \sigma_i$ where $\circ$ is horizontal composition.

Lemma 8. *K^{Fn}_{Ix} is a comparison functor from the resolution $\mathit{Free}_{\mathit{Fn}} \uparrow \dashv \downarrow U_{\mathit{Fn}}$ of functorial algebras to the resolution $\mathit{Free}_{\mathit{Ix}} \upharpoonleft \dashv \downharpoonright U_{\mathit{Ix}}$ of indexed algebras.*

Since comparison functors preserve interpretation (Lemma 5), the lemma above implies that the expressivity of functorial algebras is not greater than indexed ones. Together with the comparison functors defined earlier, we conclude that the three kinds of algebras—indexed, functorial and Eilenberg-Moore algebras—have the same expressivity for interpreting scoped operations.

Remark 4. Although the three kinds of algebras have the same expressivity in theory, they structure the interpretation of scoped operations in different ways: EM algebras impose no constraint on how the part of syntax enclosed by scopes is handled; indexed algebras demand them to be handled layer by layer but impose no coherent conditions between the layers; functorial algebras additionally force all inner layers must be handled in a uniform way by an endofunctor.

On the whole, it is a trade-off *simplicity* and *structuredness*: EM algebras are the simplest for implementation, whereas the structuredness of functorial algebras make them easier to reason about. This is another instance of the preference for structured programming over unstructured language features, in the same way as structured loops being favoured over `goto`, although they have the same expressivity in theory [13].

5 Fusion Laws of Interpretation

An advantage of the adjoint-theoretic approach to syntax and semantics is that the naturality of an adjunction directly offers *fusion laws* of interpretation that fuse a morphism after an interpretation into a single interpretation, which have proven to be a powerful tool for reasoning about and optimising programs manipulating abstract syntax [12, 21, 65, 66] and in particular handlers of algebraic effects [69, 72]. In this section, we present the fusion law for functorial algebras.

5.1 Fusion Laws of Interpretation

Recall that given any resolution $L \dashv R$ with counit ϵ of some monad $M : \mathbb{C} \to \mathbb{C}$ where $L : \mathbb{C} \to \mathbb{D}$, for any $g : A \to RD$, we have an interpretation morphism

$$\mathit{handle}_D\ g = R(\epsilon_D \cdot Lg) : MA \to RD$$

Then whenever we have a morphism in the form of $(f \cdot \mathit{handle}_D\ g)$—an interpretation followed by some morphism—the following *fusion law* allows one to fuse it into a single interpretation morphism.

Lemma 9 (Interpretation Fusion). *Assume $L \dashv R$ is a resolution of monad $M : \mathbb{C} \to \mathbb{C}$ where $L : \mathbb{C} \to \mathbb{D}$. For every $D : \mathbb{D}$, $g : A \to RD$ and $f : RD \to X$, if there is some D' and $h : D \to D'$ in $\mathbb{D}$ such that $RD' = X$ and $Rh = f$, then*

$$f \cdot \mathit{handle}_D\ g = \mathit{handle}_{D'}\ (f \cdot g) \tag{28}$$

Applying the lemma to the three resolutions of P gives us three fusion laws: for any $D : \mathbb{D}$ where $\mathbb{D} \in \{\mathtt{Ix\text{-}Alg}, \mathtt{Fn\text{-}Alg}, \mathbb{C}^P\}$, one can fuse $f \cdot \mathit{handle}_D\ g$ into a single interpretation if one can make f a $\mathbb{D}$-homomorphism. Particularly, the following is the fusion law for functorial algebras.

Corollary (Fusion Law for Functorial Algebras). *Let $\hat{\alpha_1} = \langle H, X_1, \alpha_1^G, \alpha_2^I \rangle$ be a functorial algebra (Definition 1) and $g : A \to X_1$, $f : X_1 \to X_2$ be any morphisms in $\mathbb{C}$. If there is a functorial algebra $\hat{\alpha_2} = \langle H_2, X_2, \alpha_2^G, \alpha_2^I \rangle$ and a functorial algebra morphism $\langle \sigma : H_1 \to H_2, h : X_1 \to X_2 \rangle$, then*

$$f \cdot \mathit{handle}_{\hat{\alpha_1}}\ g = \mathit{handle}_{\hat{\alpha_2}}\ (f \cdot g)$$

Example 11. Let $\hat{\alpha}$ be the functorial algebra of nondeterminism with *once* in Example 8 and $\mathit{len} : \mathit{List}\, A \to \mathbb{N}$ be the function mapping a list to its length. Then using the fusion law, $\mathit{len} \cdot \mathit{handle}_{\hat{\alpha}}\ g = \mathit{handle}_{\hat{\beta}}\ (\mathit{len} \cdot g)$ if we can find a suitable functorial algebra $\hat{\beta} : \mathtt{Fn\text{-}Alg}$ and $h : \hat{\alpha} \to \hat{\beta}$ s.t. $\downarrow \mathtt{U_{Fn}} h = \mathit{len}$. In fact, a suitable $\hat{\beta}$ is just the functorial algebra in Example 9 and $h = \langle \mathit{id}, \mathit{len} \rangle$.

Example 12. Although Piróg et al. [46] propose the adjunction (24) to interpret scoped operations with indexed algebras, their Haskell implementation is not a faithful implementation of the interpretation morphism (26), but rather a more efficient one skipping the step of transforming P to the isomorphic free indexed algebra $(\downharpoonleft \mathtt{U_{Ix}Free_{Ix}} \upharpoonright)$. However, it is previously unclear whether this implementation indeed coincides with the interpretation morphism (26) due to the discrepancy between the syntax monad P and indexed algebras.

This issue is in fact one of the original motivations for us to develop functorial algebras—a way to interpret P that directly follows the syntactic structure. Using the comparison functors to transform between indexed and functorial algebras, we can reason about Piróg et al. [46]'s implementation with functorial algebras, and its correctness can be established using fusion laws. This extended case study is in the appendices.

6 Related Work

The most closely related work is that of Piróg et al. [46] on categorical models of scoped effects. That work in turn builds on Wu et al. [70] who introduced the notion of scoped effects after identifying modularity problems with using algebraic effect handlers for catching exceptions [52]. Scoped effects have found their way into several Haskell implementations of algebraic effects and handlers [32,42,56].

Effect Handlers and Modularity Spivey [60], Moggi [44] and Wadler [67] initiated monads for modeling and programming with computational effects. Soon after, the desire arose to define complex monads by combining modular definitions of individual effects [26, 63], and monad transformers were developed to meet this need [39]. Yet, several years later, algebraic effects were proposed as an alternative more structured approach for defining and combining computational effects [22, 48, 49]. The addition of handlers [52] has made them practical for implementation and many languages and libraries have been developed since. Schrijvers et al. [58] have characterised modular handlers by means of modular carriers, and shown that they correspond to a subclass of monad transformers.

Scoped operations are generally not algebraic operations in the original design of algebraic effects [48], but as we have seen in Section 4.1, an alternative view on Eilenberg-Moore algebras of scoped operations is regarding them as handlers of *algebraic* operations of signature $\Sigma + \Gamma P$. However, the functor $\Sigma + \Gamma P$ involves the type P modelling computations, and thus it is not a valid signature of algebraic effects in the original design of effect handlers [51, 52], in which the signature of algebraic effects can only be built from some *base types* to avoid the interdependence of the denotations of signature functors and computations. In spite of that, many later implementations of effect handlers such as Eff [7], Koka [36] and Frank [40] do not impose this restriction on signature functors (at the cost that the denotational semantics involves solving recursive domain-theoretic equations), and thus scoped operations can be implemented in these languages with EM algebras as handlers.

Other variations of scoped effects have been suggested. Recently, Poulsen et al. [53] and van den Berg et al. [9] have proposed a notion of *staged* or *latent* effect, which is a variant of scoped effects, for modelling the deferred execution of computations inside lambda abstractions and similar constructs. Ahman and Pretnar [3] investigate *asynchronous effects*, and they note that interrupt handlers are in fact scoped operations. We have not yet investigated this in our framework, but it will be an interesting use case.

Abstract Syntax This work focusses on the problem of abstract syntax and semantics of programs. The practical benefit of abstract syntax is that it allows for *generic programming* in languages like Haskell that have support for, e.g. type classes, GADTs [25] and so on. As an example, Swierstra [64] showed that it is possible to modularly create compilers by formalising syntax in Haskell.

Fiore et al. [16,17] first formalise abstract syntax categorically for operations with variable binding. Subsequently, Ghani and Uustalu [18] model the abstract syntax of explicit substitutions as an initial algebra in the endofunctor category and show that it is a monad. Piróg et al. [46] and this paper use a monad P, which is a slight generalisation of the monad of explicit substitutions, to model the syntax of scoped operations. The datatype underlying P is an instance of *nested datatypes* studied by Bird and Paterson [10] and Johann and Ghani [24].

In this paper we have not treated *equations* on effectful operations, which are both theoretically and practically important. Plotkin and Power [48] show that theories of various effects with suitable equations *determine* their corresponding monads, and later Hyland et al. [22] show that certain combinations of effect theories are equivalent to monad transformers. Equations are also used for reasoning about programs with algebraic effects and handlers [34, 50, 72]. Possible ways to extend scoped effects with equations include the approach in [29] (Remark 1), the categorical framework of *equational systems* [14], second order Lawvere theories [15], and syntactic frameworks like [62].

7 Conclusion

The motivation of this work is to develop a structured approach to the syntax and semantics of scoped operations. We believe our proposal, functorial algebras, is at a sweet spot in the trade-off between structuredness and simplicity, allowing practical examples of scoped operations to be programmed and reasoned about naturally, and implementable in modern functional languages such as Haskell and OCaml. We put our model and two other models for interpreting scoped effects in the same categorical framework, and we showed that they have equivalent expressivity for interpreting scoped effects, although they form non-equivalent categories. The uniform theoretical framework also induces fusion laws of interpretation in a straightforward way.

There are two strains of work that should be pursued from here. The first one would be investigating ways to compose algebras of scoped operations. The second one would be the design of a language supporting handlers of scoped operations natively and its type system and operational semantics.

Acknowledgements

This work is supported by EPSRC grant number EP/S028129/1 on 'Scoped Contextual Operations and Effects', by FWO project G095917N, and KU Leuven project C14/20/079. The authors would like to thank the anonymous reviewers for their constructive feedback.

References

1. Adámek, J., Rosicky, J.: Locally Presentable and Accessible Categories. London Mathematical Society Lecture Note Series, Cambridge University Press (1994). https://doi.org/10.1017/CBO9780511600579

2. Adámek, J.: Free algebras and automata realizations in the language of categories. Commentationes Mathematicae Universitatis Carolinae **015**(4), 589–602 (1974), http://eudml.org/doc/16649
3. Ahman, D., Pretnar, M.: Asynchronous effects. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). https://doi.org/10.1145/3434305
4. Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Categorical fixed point calculus. In: Pitt, D., Rydeheard, D.E., Johnstone, P. (eds.) Category Theory and Computer Science. pp. 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60164-3_25
5. Barr, M.: Coequalizers and free triples. Mathematische Zeitschrift (1970)
6. Barr, M., Wells, C.: Category theory for computing science, vol. 1. Prentice Hall New York (1990)
7. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. Logical Methods in Computer Science **10**(4) (Dec 2014). https://doi.org/10.2168/lmcs-10(4:9)2014
8. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebraic Methods Program. **84**(1), 108–123 (2015). https://doi.org/10.1016/j.jlamp.2014.02.001
9. van den Berg, B., Schrijvers, T., Bach-Poulsen, C., Wu, N.: Latent effects for reusable language components: Extended version (2021), https://arxiv.org/abs/2108.11155
10. Bird, R.S., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects Comput. (1999). https://doi.org/10.1007/s001650050047
11. Claessen, K.: A poor man's concurrency monad. Journal of Functional Programming **9**(3), 313–323 (1999). https://doi.org/10.1017/S0956796899003342
12. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. SIGPLAN Not. **42**(9), 315–326 (Oct 2007). https://doi.org/10.1145/1291220.1291199
13. Dijkstra, E.W.: Letters to the editor: Go to statement considered harmful. Commun. ACM **11**(3), 147–148 (Mar 1968). https://doi.org/10.1145/362929.362947
14. Fiore, M., Hur, C.K.: On the construction of free algebras for equational systems. Theoretical Computer Science **410**(18), 1704–1729 (2009). https://doi.org/https://doi.org/10.1016/j.tcs.2008.12.052, automata, Languages and Programming (ICALP 2007)
15. Fiore, M., Mahmoud, O.: Second-order algebraic theories. In: Hliněný, P., Kučera, A. (eds.) Mathematical Foundations of Computer Science 2010. pp. 368–380. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_33
16. Fiore, M.P., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999 (1999). https://doi.org/10.1109/LICS.1999.782615
17. Fiore, M.P., Turi, D.: Semantics of name and value passing. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings (2001). https://doi.org/10.1109/LICS.2001.932486
18. Ghani, N., Uustalu, T.: Explicit substitutions and higher-order syntax. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding. p. 1–7. MERLIN '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/976571.976580

19. Hillerström, D., Lindley, S.: Shallow Effect Handlers. Lecture Notes in Computer Science **11275 LNCS**, 415–435 (2018). https://doi.org/10.1007/978-3-030-02768-1_22
20. Hinze, R.: Prological features in a functional setting — axioms and implementations. In: Sato, M., Toyama, Y. (eds.) Proceedings of the Third Fuji International Symposium on Functional and Logic Programming (FLOPS '98). pp. 98–122. World Scientific, Singapore, New Jersey, London, Hong Kong (apr 1998)
21. Hinze, R., Harper, T., James, D.W.H.: Theory and practice of fusion. In: Hage, J., Morazán, M.T. (eds.) Implementation and Application of Functional Languages. pp. 19–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24276-2_2
22. Hyland, M., Plotkin, G., Power, J.: Combining effects: Sum and tensor. Theor. Comput. Sci. **357**(1), 70–99 (Jul 2006). https://doi.org/10.1016/j.tcs.2006.03.013
23. Hyland, M., Power, J.: The category theoretic understanding of universal algebra: Lawvere theories and monads. Electronic Notes in Theoretical Computer Science **172**, 437–458 (2007). https://doi.org/10.1016/j.entcs.2007.02.019, computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin
24. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Typed Lambda Calculi and Applications, TLCA. Lecture Notes in Computer Science, Springer (2007). https://doi.org/10.1007/978-3-540-73228-0_16
25. Johann, P., Ghani, N.: Foundations for structured programming with gadts. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 297–308. ACM (2008). https://doi.org/10.1145/1328438.1328475
26. Jones, M.P., Duponcheel, L.: Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA (December 1993), http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf
27. Kammar, O., Lindley, S., Oury, N.: Handlers in action. SIGPLAN Not. **48**(9), 145–158 (Sep 2013). https://doi.org/10.1145/2544174.2500590
28. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimisations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 349–360. POPL '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103698
29. Kelly, G.M.: Structures defined by finite limits in the enriched context, i. Cahiers de Topologie et Géométrie Différentielle Catégoriques **23**(1), 3–42 (1982), http://www.numdam.org/item/CTGDC_1982__23_1_3_0/
30. Kelly, G., Power, A.: Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. Journal of Pure and Applied Algebra **89**(1), 163–179 (1993). https://doi.org/10.1016/0022-4049(93)90092-8
31. Kidney, D.O., Wu, N.: Algebras for weighted search. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473577
32. King, A.: eff – screaming fast extensible effects for less (2019), https://github.com/hasura/eff
33. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. p. 94–105. Haskell '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2804302.2804319

34. Kiselyov, O., Mu, S.C., Sabry, A.: Not by equations alone: Reasoning with extensible effects. Journal of Functional Programming **31**, e2 (2021). https://doi.org/10.1017/S0956796820000271
35. Lambek, J., Scott, P.J.: Introduction to Higher Order Categorical Logic. Cambridge University Press, USA (1986)
36. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 486–499. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009872
37. Levy, P.B.: Adjunction models for call-by-push-value with stacks. Electronic Notes in Theoretical Computer Science **69**, 248–271 (2003). https://doi.org/https://doi.org/10.1016/S1571-0661(04)80568-1, cTCS'02, Category Theory and Computer Science
38. Levy, P.B.: Call-by-push-value: A Functional/Imperative Synthesis, vol. 2. Springer Netherlands (2003). https://doi.org/10.1007/978-94-007-0954-6
39. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 333–343. POPL '95, ACM (1995). https://doi.org/10.1145/199448.199528
40. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 500–514. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009897
41. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics, Springer, Berlin (1998)
42. Maguire, S.: polysemy: Higher-order, low-boilerplate free monads (2019), https://hackage.haskell.org/package/polysemy
43. Matthes, R., Uustalu, T.: Substitution in non-wellfounded syntax with variable binding. Theoretical Computer Science **327**(1), 155–174 (2004). https://doi.org/https://doi.org/10.1016/j.tcs.2004.07.025, selected Papers of CMCS '03
44. Moggi, E.: An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Edinburgh University, Department of Computer Science (June 1989)
45. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55 – 92 (1991). https://doi.org/https://doi.org/10.1016/0890-5401(91)90052-4, selections from 1989 IEEE Symposium on Logic in Computer Science
46. Piróg, M., Schrijvers, T., Wu, N., Jaskelioff, M.: Syntax and semantics for operations with scopes. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. p. 809–818. LICS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209166
47. Piróg, M., Gibbons, J.: Tracing monadic computations and representing effects. Electronic Proceedings in Theoretical Computer Science **76**, 90–111 (Feb 2012). https://doi.org/10.4204/eptcs.76.8
48. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) Foundations of Software Science and Computation Structures, 5th International Conference. pp. 342–356. FOSSACS 2002, Springer (2002). https://doi.org/10.1007/3-540-45931-6_24
49. Plotkin, G., Power, J.: Algebraic operations and generic effects. Applied Categorical Structures **11**(1), 69–94 (2003). https://doi.org/10.1023/A:1023064908962

50. Plotkin, G., Pretnar, M.: A logic for algebraic effects. In: 2008 23rd Annual IEEE Symposium on Logic in Computer Science. pp. 118–129 (2008). `https://doi.org/10.1109/LICS.2008.45`
51. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) Programming Languages and Systems. pp. 80–94. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). `https://doi.org/10.1007/978-3-642-00590-9_7`
52. Plotkin, G., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (Dec 2013). `https://doi.org/10.2168/lmcs-9(4:23)2013`
53. Poulsen, C.B., van der Rest, C., Schrijvers, T.: Staged effects and handlers for modular languages with abstraction. In: Workshop on Partial Evaluation and Program Manipulation (PEPM) (2021), `https://casvdrest.github.io/staged-effects.agda/pepm21.pdf`
54. Power, A.J.: Enriched Lawvere theories. Theory and Applications of Categories **6**(7), 83–93 (1999)
55. Riehl, E.: Category Theory in Context. Aurora: Dover Modern Math Originals, Dover Publications (2017)
56. Rix, R., Thomson, P., Wu, N., Schrijvers, T.: fused-effects: A fast, flexible, fused effect system (2018), `https://hackage.haskell.org/package/fused-effects`
57. Robinson, E.: Variations on algebra: Monadicity and generalisations of equational theories. Form. Asp. Comput. **13**(3–5), 308–326 (Jul 2002). `https://doi.org/10.1007/s001650200014`
58. Schrijvers, T., Piróg, M., Wu, N., Jaskelioff, M.: Monad transformers and modular algebraic effects: what binds them together. In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. pp. 98–113 (2019). `https://doi.org/10.1145/3331545.3342595`
59. Spivey, J.M.: Algebras for combinatorial search. Journal of Functional Programming **19**(3-4), 469–487 (2009). `https://doi.org/10.1017/S0956796809007321`
60. Spivey, M.: A functional theory of exceptions. Science of Computer Programming **14**(1), 25–42 (1990). `https://doi.org/10.1016/0167-6423(90)90056-J`
61. Stark, I.: Free-algebra models for the π-calculus. Theoretical Computer Science **390**(2), 248–270 (2008). `https://doi.org/10.1016/j.tcs.2007.09.024`
62. Staton, S.: Instances of computational effects: An algebraic perspective. In: 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 519–519 (2013). `https://doi.org/10.1109/LICS.2013.58`
63. Steele, Jr., G.L.: Building interpreters by composing monads. In: Boehm, H., Lang, B., Yellin, D.M. (eds.) Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 472–492. POPL '94, ACM (1994). `https://doi.org/10.1145/174675.178068`
64. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). `https://doi.org/10.1017/S0956796808006758`
65. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. Association for Computing Machinery, New York, NY, USA (1995). `https://doi.org/10.1145/224164.224221`
66. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (Jan 1988). `https://doi.org/10.1016/0304-3975(90)90147-A`
67. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78. LFP '90, ACM (1990). `https://doi.org/10.1145/91556.91592`

68. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. p. 24–52. Springer-Verlag, Berlin, Heidelberg (1995). `https://doi.org/10.5555/647698.734146`
69. Wu, N., Schrijvers, T.: Fusion for free. In: Hinze, R., Voigtländer, J. (eds.) Mathematics of Program Construction. pp. 302–322. Springer International Publishing, Cham (2015). `https://doi.org/978-3-319-19797-5_15`
70. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. p. 1–12. Haskell '14, Association for Computing Machinery, New York, NY, USA (2014). `https://doi.org/10.1145/2633357.2633358`
71. Yang, Z., Paviotti, M., Wu, N., van den Berg, B., Schrijvers, T.: Structured handling of scoped effects: Extended version (2022), `https://arxiv.org/abs/2201.10287`
72. Yang, Z., Wu, N.: Reasoning about effect interaction by fusion. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). `https://doi.org/10.1145/3473578`

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (`http://creativecommons.org/licenses/by/4.0/`), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.