Scoped Effects and Their Algebras

ZHIXUAN YANG, Imperial College London, United Kingdom
MARCO PAVIOTTI, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom
BIRTHE VAN DEN BERG, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium

Algebraic effects are not ideal for modularly modelling effect operations that delimit a scope. Two recent proposals for scoped effects aim to address this shortcoming, one that is more ad-hoc and suited for practical implementation and the other theoretically founded on a free monad in an indexed category, but less convenient for implementation purposes.

In this paper we aim to provide the best of both worlds, a theoretically-founded model of scoped effects that is convenient for implementation. In fact, we present two alternative models based on alternative adjunctions that give rise to alternative implementations. The first is a simple, less structured approach based on Eilenberg-Moore algebras that shows how scoped effects can be encoded in existing algebraic effects language. The second, which we consider to be the sweet spot between between ease of implementation and provided structure, is based on functorial algebras.

Using comparison functors we show that these two novel approaches are equivalent to one another and the earlier indexed approach. We exploit this fact with the fusion rules of the different approached to construct hybrid folds that mix the program syntax of one approach with the algebras of another.

Finally, to demonstrate the practical implementability of functorial algebras we provide a range of examples in both Haskell and OCaml.

ACM Reference Format:
Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2020. Scoped Effects and Their Algebras. *Proc. ACM Program. Lang.* 1, ICFP, Article 42 (January 2020), 28 pages.

1 INTRODUCTION

For a long time monads [Moggi 1991] 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*, is gaining more traction. A big breakthrough has been the introduction of *handlers*, which has made algebraic effects suitable for programming and has led to numerous dedicated languages and libraries.

In comparison to monads, algebraic effects provide a more structured approach for defining and composing effects, which explains much of their appeal. A disadvantage of algebraic effects is that they are less expressive; not all effects can be easily expressed or composed within its confines. Notably, Wu et al. [2014] identified the class of *scoped* effects that do not fit the mold. Operations like *catch* for exception handling or *once* for restricting nondeterminism are not conventional algebraic operations because they are not atomic; instead they delimit a computation within their scope. These operations are also not adequately expressed as handlers because this limits their compositionality.

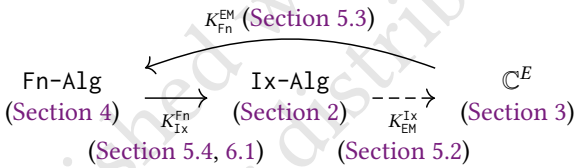
Authors' addresses: Zhixuan Yang, s.yang20@imperial.ac.uk, Department of Computing, Imperial College London, United Kingdom; Marco Paviotti, m.paviotti@imperial.ac.uk, Department of Computing, Imperial College London, United Kingdom; Nicolas Wu, n.wu@imperial.ac.uk, Department of Computing, Imperial College London, United Kingdom; Birthe van den Berg, birthe.vandenberg@kuleuven.be, KU Leuven, Department of Computer Science, Belgium; Tom Schrijvers, tom.schrijvers@kuleuven.be, KU Leuven, Department of Computer Science, Belgium.

2020. 2475-1421/2020/1-ART42 $15.00
https://doi.org/

To remedy the situation, Wu et al. [2014] proposed a practical, but ad-hoc generalization of algebraic effects that encompasses scoped effects. Their approach has been adopted by several algebraic effects libraries. More recently, Piróg et al. [2018] sought to put this ad-hoc approach for scoped effects on the same formal footing as algebraic effects. Indeed, they wished to derive the syntax of programs with scoped effects and its handlers from a free monad construction and its associated recursion scheme. First they sought such a construction on the base category of sets or types or on the category of endofunctors thereof, because these lead to a relatively straightforward implementation in many programming languages. Yet, in the end they had to settle for a construction on a level-indexed category, which is not ideal for widespread implementation as it requires support for languages with dependent typing, in at least a limited form like GADTs [Johann and Ghani 2008]. Acknowledging this downside, they have partly compensated for it with an ad-hoc hybrid recursion scheme that requires indexing for the handlers, but not for the program syntax.

This paper revisits the challenge of scoped effects. We present a new characterisation that is both principled and formally grounded like that of Piróg et al. [2018]. At the same time our approach can be practically implemented without the need for dependent types or GADTs, making it available for a wider range of programming languages. The chief insight of our paper is to demonstrate that there are three ways of interpreting scoped operations: indexed algebras, Eilenberg-Moore algebras, and functorial algebras. As we shall see, although equivalent in expressivity, these different algebras come with different trade-offs.

The following diagram summarises the more technical results of this paper, where there are three ways of interpreting scoped operations, each corresponding to a category:



Syntax trees with scoped operations are expressed by a monad $E : \mathbb{C} \to \mathbb{C}$. The three interpretations of these operations are in terms of three categories that are connected to $\mathbb{C}$ by an adjunction that gives rise to the monad $E : \mathbb{C} \to \mathbb{C}$. The indexed algebras inhabit the `Ix-Alg` category (Section 2), Eilenberg-Moore algebras inhabit the category $\mathbb{C}^E$ (Section 3), and the functorial algebras inhabit the `Fn-Alg` category (Section 4).

The Eilenberg-Moore category $\mathbb{C}^E$ is built directly from E, and it has the well-known property that there is a unique functor to it from all other categories that give rise to E through an adjunction (Section 5.2). To establish that the three representations can be translated into one another, we first show that there is a functor $K^{\mathtt{EM}}_{\mathtt{Fn}}$ that translates Eilenberg-Moore algebras to functorial algebras (Section 5.3). Having established this, we close the circle by showing that the functorial algebras of `Fn-Alg` can be translated into an indexed algebra by a functor $K^{\mathtt{Fn}}_{\mathtt{Ix}}$ (Section 5.4). Crucially, we show that the functors $K^{\mathtt{Fn}}_{\mathtt{Ix}}$, $K^{\mathtt{Ix}}_{\mathtt{EM}}$ and $K^{\mathtt{EM}}_{\mathtt{Fn}}$ *preserve interpretation*, and thus the three kinds of algebras have equal expressivity for interpreting scoped operations.

We approach this material with the background to the *indexed algebras* of Piróg et al. [2018] (Section 2), and then the main contributions of this paper are:

- We show that, by relinquishing structural recursion but not expressivity, we can use *Eilenberg-Moore algebras*: these can be implemented in languages with general recursion and higher-order functions (Section 3).
- We present a construction of scoped effects based on *functorial algebras*: these do not relinquish structural recursion, and are more readily expressed in languages without GADTs or dependent types (Section 4).

- By constructing functors between their respective categories that preserve interpretation we show that, in terms of expressive power for interpreting scoped operations, Eilenberg-Moore algebras, functorial algebras and indexed algebras are equal (Section 5).
- We present the *fusion laws* for interpreting scoped operations, which allows us to establish that the *hybrid fold* recursion scheme of Piróg et al. [2018] coincides with interpreting with indexed algebras, putting it on solid formal footing (Section 6).
- Throughout the paper we demonstrate our techniques with a number of examples using an implementation in Haskell and in OCaml.

Finally, we discuss related work (Section 7) and conclude (Section 8).

2 SYNTAX AND SEMANTICS OF SCOPED OPERATIONS

In this section we introduce the syntax and semantics for operations with scopes. To provide intuitions, we start with several programming examples of nondeterministic search with scoped operations and indexed algebras in Section 2.1, and then we show the formal theory underlying scoped operations in Section 2.2.

2.1 Working With Scoped Operations

The starting point for scoped operations is that they work with two functors, Σ and Γ, where Σ is the signature of ordinary algebraic effects, and Γ is the signature of scoped effects.

$$\textbf{data}\ \mathit{Prog}\ \Sigma\ \Gamma\ a = \mathit{Return}\ a \mid \mathit{Call}\ (\Sigma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)) \mid \mathit{Enter}\ (\Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)))$$

This datatype can be used to represent programs that either return a pure value x with $\mathit{Return}\ x$, call an operation op that has a continuation k with $\mathit{Call}\ (op\ k)$, or enter a scoped computation $scope$ with $\mathit{Enter}\ (scope\ k)$.

For instance, the effect of nondeterministic choice can be modelled using the *Choice* signature, which is a functor in a:

$$\textbf{data}\ \mathit{Choice}\ a = \mathit{Fail} \mid \mathit{Or}\ a\ a\ \textbf{deriving}\ \mathit{Functor}$$

Operation *Fail* indicates that there is no choice, and $\mathit{Or}\ x\ y$ indicates that either x or y is available.

There is a monad instance of *Prog* that allows us to compose programs sequentially:

$$\begin{aligned}&\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Monad}\ (\mathit{Prog}\ f\ g)\ \textbf{where}\\ &\quad \mathit{return} = \mathit{Return}\\ &\quad \mathit{Return}\ x \gg\!= f = f\ x\\ &\quad \mathit{Call}\ op \gg\!= f\ \ = \mathit{Call}\ (\mathit{fmap}\ (\gg\!= f)\ op)\\ &\quad \mathit{Enter}\ sc \gg\!= f\ \ = \mathit{Enter}\ (\mathit{fmap}\ (\mathit{fmap}\ (\gg\!= f))\ sc)\end{aligned}$$

With the help of the monadic structure and smart constructors $\mathit{fail} = \mathit{Call}\ \mathit{Fail}$ and $\mathit{or}\ x\ y = \mathit{Call}\ (\mathit{Or}\ x\ y)$, we can write programs such as the following:

$$\begin{aligned}&\mathit{select} :: \mathit{Functor}\ \Gamma \Rightarrow [a] \to \mathit{Prog}\ \mathit{Choice}\ \Gamma\ (a, [a])\\ &\mathit{select}\ [\,]\ \ \ \ \ \ = \mathit{fail}\\ &\mathit{select}\ (x : xs) = \mathit{return}\ (x, xs)\ `\mathit{or}`\ \textbf{do}\ \{(y, ys) \leftarrow \mathit{select}\ xs; \mathit{return}\ (y, x : ys)\}\end{aligned}$$

This program takes a list xs and returns all pairs of the form (z, zs), where $z : zs$ is some permutation of xs. This can be use to select an element without replacement from xs, leaving the remaining unselected elements in zs.

To produce all the permutations in a list, we can define the *perm* program:

$$\mathit{perm} :: \mathit{Functor}\ \Gamma \Rightarrow [a] \to \mathit{Prog}\ \mathit{Choice}\ \Gamma\ [a]$$

$$
\begin{aligned}
&\mathit{perm}\ [\,] = \mathit{return}\ [\,]\\
&\mathit{perm}\ \mathit{xs} = \mathbf{do}\ (y, \mathit{ys}) \leftarrow \mathit{select}\ \mathit{xs}; \mathit{zs} \leftarrow \mathit{perm}\ \mathit{ys}; \mathit{return}\ (y : \mathit{zs})
\end{aligned}
$$

This uses *select* to pick out the first element y in a permutation, and adds this to the result of permuting the rest of *ys*. So far, this is is essentially the same as programming within the list monad.

The beauty of algebraic effects stated in this way is that a semantics to this program can be easily given by stating how *Return* and the operations *Fail* and *Or* should be interpreted. Functions that interpret *Return* are called *generators*, and functions that interpret operations are called *algebras*.

If using the list monad is desired then the following functions would be enough information to completely determine a function $\mathit{list} :: \mathit{Functor}\ \Gamma \Rightarrow \mathit{Prog}\ \mathit{Choice}\ \Gamma\ a \to [a]$, where $\mathit{list}\ (\mathit{perm}\ \mathit{xs})$ produces all permutations of *xs*.

$$
\begin{aligned}
&\mathit{return}_{\mathit{list}} :: a \to [a] && \mathit{call}_{\mathit{list}} :: \mathit{Choice}\ [a] \to [a]\\
&\mathit{return}_{\mathit{list}}\ x = [x] && \mathit{call}_{\mathit{list}}\ \mathit{Fail} = [\,]\\
& && \mathit{call}_{\mathit{list}}\ (\mathit{Or}\ \mathit{xs}\ \mathit{ys}) = \mathit{xs} + \mathit{ys}
\end{aligned}
$$

With suitable engineering, these functions are used to replace all occurrences of *Return*, *Fail* and *Or* with computations produces all solutions to a nondeterministic computation. The function *list*, and others like it that use generators and algebras to interpret a program are called *handlers*.

Handlers allow different interpretations of a program. For instance, here is another generator and algebra for the same trees:

$$
\begin{aligned}
&\mathit{return}_{\mathit{rand}} :: a \to [\mathit{Bool}] \to \mathit{Maybe}\ a && \mathit{call}_{\mathit{rand}} :: \mathit{Choice}\ ([\mathit{Bool}] \to \mathit{Maybe}\ a) \to ([\mathit{Bool}] \to \mathit{Maybe}\ a)\\
&\mathit{return}_{\mathit{rand}}\ x\ \mathit{bs} = \mathit{Just}\ x && \mathit{call}_{\mathit{rand}}\ \mathit{Fail}\ \mathit{bs} = \mathit{Nothing}\\
& && \mathit{call}_{\mathit{rand}}\ (\mathit{Or}\ \mathit{fxs}\ \mathit{fys})\ (b : \mathit{bs}) = \mathbf{if}\ b\ \mathbf{then}\ (\mathit{fxs}\ \mathit{bs})\ \mathbf{else}\ (\mathit{fys}\ \mathit{bs})
\end{aligned}
$$

These functions can be used to define $\mathit{rand} :: \mathit{Functor}\ \Gamma \Rightarrow \mathit{Prog}\ \mathit{Choice}\ \Gamma\ a \to [\mathit{Bool}] \to \mathit{Maybe}\ a$, where this time $\mathit{rand}\ (\mathit{perm}\ \mathit{xs})\ \mathit{bs}$ will return a random permutation of *xs* when given a random stream of booleans *bs*.

So far, we have focused on *algebraic* effects, which are given by algebras that distribute through bind. However, not all programs can be interpreted by algebras. For instance, searching for solutions in a nondeterministic program is rarely done by enumerating all the possibilities: heuristics are usually employed to reduce the search space. One such heuristic is to use the *once* operation, which returns only the first solution of the program in its argument. Piróg et al. [2018] demonstrated that this is not an algebraic operation, and is instead a *scoped* operation. To see this, for example, the following program is intended to return either 0 or 1:

$$
\begin{aligned}
&\mathbf{do}\ \{x \leftarrow \mathit{once}\ (\mathit{return}\ 0\ `\mathit{or}`\ \mathit{return}\ 1); \mathit{return}\ x\ `\mathit{or}`\ \mathit{return}\ (x+1)\}\\
&= \mathbf{do}\ \{x \leftarrow \mathit{return}\ 0; \mathit{return}\ x\ `\mathit{or}`\ \mathit{return}\ (x+1)\}\\
&= \mathit{return}\ 0\ `\mathit{or}`\ \mathit{return}\ (0+1)
\end{aligned}
$$

but if *once* is an algebraic operation [Plotkin and Power 2002], then

$$
\begin{aligned}
&\mathbf{do}\ \{x \leftarrow \mathit{once}\ (\mathit{return}\ 0\ `\mathit{or}`\ \mathit{return}\ 1); \mathit{return}\ x\ `\mathit{or}`\ \mathit{return}\ (x+1)\}\\
&= \mathit{once}\ ((\mathit{return}\ 0\ `\mathit{or}`\ \mathit{return}\ (0+1))\ `\mathit{or}`\ (\mathit{return}\ 1\ `\mathit{or}`\ \mathit{return}\ (1+1)))\\
&= \mathit{return}\ 0
\end{aligned}
$$

Thus operations like *once* that delimit a scope but are not algebraic shall be treated differently from algebraic ones. We represent such scoped operations by the Γ signature:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Once}\ a = \mathit{Once}\ a\ \mathbf{deriving}\ \mathit{Functor}\\
&\mathit{once} :: \mathit{Functor}\ \Sigma \Rightarrow \mathit{Prog}\ \Sigma\ \mathit{Once}\ a \to \mathit{Prog}\ \Sigma\ \mathit{Once}\ a
\end{aligned}
$$

```
data Nat = Zero | Nat + 1
data IxAlg Σ Γ a =
  IxAlg { action :: ∀n. Σ (a n) → a n, demote :: ∀n. Γ (a (n + 1)) → a n, promote :: ∀n. a n → a (n + 1) }
hfold :: (Functor f, Functor g) ⇒ IxAlg f g a → ∀n. Prog f g (a n) → a n
hfold ixAlg (Return x)    = x
hfold ixAlg (Call op)     = action ixAlg (fmap (hfold ixAlg) op)
hfold ixAlg (Enter scope) = demote ixAlg (fmap (hfold ixAlg · fmap (promote ixAlg · hfold ixAlg)) scope)
```

Fig. 1. The hybrid fold for interpreting monad E with indexed algebras [Piróg et al. 2018]

$$once\ p = Enter\ (Once\ (fmap\ Return\ p))$$

The type of $Enter :: \Gamma\ (Prog\ \Sigma\ \Gamma\ (Prog\ \Sigma\ \Gamma\ a)) \to Prog\ \Sigma\ \Gamma\ a$ shows that this is no ordinary operation. Its argument is a program whose leaves are themselves programs: each invocation of *once* creates a new nested level of interpretation.

For our example of nondeterminism, if we wish to collect all the solutions using semantics like that of *list*, interpreting the part of the program **do** $\{x \leftarrow once\ p; k\}$ inside the scope of *once* will return a value of type $[[a]]$. This nested structure must then be collapsed to extract a final list. We will write $[[a]]^n$ to represent a list with n levels of extra nesting. That is, $[[a]]^0 = [a]$, $[[a]]^1 = [[a]]$, $[[a]]^2 = [[[a]]]$ and so on.

The algebras for working with algebraic operations remain essentially the same, but the scoped operations require special treatment to deal with the nested structure. Interpreting such operations requires functions that *promote* and *demote* values from scoped contexts.

$$promote_{once} :: \forall(n :: Nat).\ [[a]]^n \to [[a]]^{n+1} \qquad demote_{once} :: \forall(n :: Nat).\ Once\ [[a]]^{n+1} \to [[a]]^n$$
$$promote_{once}\ xs = [xs] \qquad demote_{once}\ (Once\ [\,]) = [\,]$$
$$demote_{once}\ (Once\ (xs : xss)) = xs$$

The $promote_{once}$ function adds an extra level of nesting, reflecting the fact that we are entering one extra layer of scope. It does so by simply returning a list containing the given list. The $demote_{once}$ function must remove one level of nesting, and it does so by picking the first nested list if it exists.

Together with the functions $return_{list}$ and $call_{list}$, it is possible to write a handler for scoped operations of type $Prog\ Choice\ Once\ a \to [a]$ that extracts all possible solutions, modulo those pruned by the *once* heuristic. Note that this implementation is only made possible in the presence of type-level programming because of the natural number indices in the type $[[a]]^n$. In Haskell, this requires extensions such as GADTs and judicial use of types to encode natural numbers, or the `DataKinds` extension, that allows values to be treated as types.

Scoped operations can be interpreted by *hfold*, a function we call a *hybrid fold* for reasons we discuss later. The implementation given by Piróg et al. [2018] is in Figure 1.

2.2 Foundations of Scoped Operations

In the rest of this section we introduce the categorical foundation underlying scoped operations and indexed algebras introduced by [Piróg et al. 2018]. As we will see in later sections, the categorical formulation not only provides an elegant unifying framework to describe algebras of scoped operations, but also equips us with useful tools to construct, compare and even optimise them, and indeed some of our proofs are made much easier by some elementary results in category theory. Thus we believe that our effort of going abstract here is worthwhile.

Throughout this paper, we use boldface letters such as $\mathbb{C}$ and $\mathbb{D}$ for variables of categories and typewriter font for specific categories such as `Set` and `Ix-Alg`. Functors and objects are denoted by capitalised letters such as F, K, G, X, A in general, and functors representing signatures are always denoted by Greek letters Γ and Σ, and some specific functors are denoted by typewriter font such as `Free` and `Ix`. Morphisms and natural transformations are denoted by uncapitalised letters such as f, g and h and Greek letters such as α, β and τ.

For any two categories $\mathbb{C}$ and $\mathbb{D}$, we write $\mathbb{C} \times \mathbb{D}$ for their *coproduct category*, whose objects are denoted by $\langle X, Y\rangle$ where $X : \mathbb{C}$ and $Y : \mathbb{D}$ and morphisms are also denoted by $\langle f, g\rangle$ where $f : X \to X'$ and $g : Y \to Y'$. The *functor category* from $\mathbb{C}$ to $\mathbb{D}$ is denoted by $\mathbb{D}^{\mathbb{C}}$. We also use $X \times Y$ for the *product* of two objects in a category and $X + Y$ for their *coproduct*. For coproducts, the injection morphisms to a coproduct are denoted by $\iota_n : X \to X_1 + \cdots + X_n$, and the universal morphism out of the coproduct is denoted by $[f, g] : X + Y \to Z$ for $f : X \to Z$ and $g : Y \to Z$. Lastly, the (carrier of) *initial algebra* of an endofunctor $G : \mathbb{C} \to \mathbb{C}$ is denoted by μG or $\mu Y.\, GY$, and the *catamorphism* from it to G-algebra $\alpha : GX \to X$ is denoted by $(\!|\alpha|\!)$.

2.2.1 Syntax of Scoped Operations. We assume a category $\mathbb{C}$ to be the *base category* where programs lives in. Category $\mathbb{C}$ is assumed to have finite products and coproducts and for simplicity to have enough initial algebras to model the syntax of programs. A signature Σ is a finite collection of operations $\{\mathsf{op}_i\}_{1 \leqslant i \leqslant n}$, where each op_i has arity $\sigma(i) \in \mathbb{N}$

$$\mathsf{op}_i : \underbrace{X \times \cdots \times X}_{\sigma(i)} \to X$$

and thus can be represented by endofunctors $\Sigma : \mathbb{C} \to \mathbb{C}$ such that $\Sigma X = X^{\sigma(1)} + \cdots + X^{\sigma(n)}$ where X^i denotes the i-fold product of X. Throughout this paper, we fix two signature functors Σ and Γ for algebraic and scoped operations respectively.

As we have seen in the programming examples, the syntax of programs involving scoped operations are modelled by datatype *Prog f g*, which is a *nested datatype* [Bird and Paterson 1999; Johann and Ghani 2007]. Following Piróg et al. [2018], we model it by an initial algebra in the endofunctor category $\mathbb{C}^{\mathbb{C}}$.

Definition 2.1 (Syntax Endofunctor)**.** Given signature functors Σ and Γ of algebraic and scoped operations respectively, letting functor $G : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}^{\mathbb{C}}$ be $GH = Id + \Sigma H + \Gamma HH$, then the *syntax endofunctor* $E : \mathbb{C}^{\mathbb{C}}$ is defined to be the initial algebra μG, which models syntax trees of programs involving operations from Σ and Γ.

For modelling sequential composition of programs, Piróg et al. [2018] showed that the syntax endofunctor E can be equipped with a monadic structure based on the *free monad* induced by the signature functor $(\Sigma + \Gamma E)$. We brief describe the construction below.

Given any endofunctor $F : \mathbb{C}^{\mathbb{C}}$, an F-algebra is a tuple $\langle X : \mathbb{C}, \alpha : FX \to X\rangle$ where the object X is called the *carrier* and the morphism α is called the *structure map* of the algebra, which represents the *operations* on the carrier. An *F-algebra homomorphism* from $\langle X, \alpha\rangle$ to $\langle X', \alpha'\rangle$ is a morphism $f : X \to X'$ in $\mathbb{C}$ such that it preserves the operations on the carrier X, that is $f \cdot \alpha = \alpha' \cdot Ff$. The category of F-algebras is denoted by F-`Alg`.

The category F-`Alg` is connected to its base category $\mathbb{C}$ through the *free-forgetful adjunction*:

$$F\text{-}\mathtt{Alg} \underset{\mathsf{U}_F}{\overset{\mathsf{Free}_F}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C} \tag{1}$$

where $\mathsf{U} : F$-`Alg` $\to \mathbb{C}$ is the obvious "forgetful" functor forgetting the structure map and returning the carrier. Its *left adjoint* takes an object A in $\mathbb{C}$ and gives an object in F-`Alg` called the *free*

F-algebra generated by A. The carrier of the free algebra is denoted by F^*A and its structure map is denoted by $\mathsf{op}_A : FF^*A \to F^*A$. The monad $\mathsf{U}_F\mathsf{Free}_F$ induced by this adjunction is also denoted by $F^* : \mathbb{C} \to \mathbb{C}$.

Now back to scoped operations, as noted by Piróg et al. [2018], the syntax endofunctor E is isomorphic to the free monad $(\Sigma + \Gamma E)^*$ as endofunctors. Thus we can equip E with the same monadic structure as $(\Sigma + \Gamma E)^*$, which is precisely *Prog*, shown in the beginning of Section 2.1.

2.2.2 Semantics of Scoped Operations. For algebraic operations, Plotkin and Pretnar [2013] introduced the notion of *handlers* to give semantics to syntax trees of programs. Handlers are essentially Σ-algebras for a signature functor Σ, and thus the adjunction $\mathsf{Free}_\Sigma \dashv \mathsf{U}_\Sigma$ underpins both the syntax and semantics of programs with algebraic operations: syntax is modelled by the monad induced by the adjunction $\Sigma^* = \mathsf{U}_\Sigma\mathsf{Free}_\Sigma$, and semantics is given by the objects in Σ-`Alg`.

This approach is called an *adjoint-theoretic approach* to syntax and semantics by Piróg et al. [2018], and they extended it to incorporate scoped operations that are not algebraic. In particular, they defined a category `Ix-Alg` of algebras called *indexed algebras* of scoped operations, which are intuitively natural for interpreting programs with scoped operations, and importantly, they showed that there is an adjunction between `Ix-Alg` and the base category $\mathbb{C}$ that induces the monad E modelling syntax of programs with scoped operations. Thus this adjunction mirrors the role played by the free-forgetful adjunction for algebraic operations and handlers.

Specifically, their adjunction for indexed algebras is constructed by composing two adjunctions:

$$\mathtt{Ix\text{-}Alg} \underset{\mathsf{U}_{\mathsf{Ix}}}{\overset{\mathsf{Free}_{\mathsf{Ix}}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C}^{|\mathbb{N}|} \underset{\downarrow}{\overset{\uparrow}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C} \tag{2}$$

Here $\mathbb{C}^{|\mathbb{N}|}$ 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 between the levels). An endofunctor $\mathtt{Ix} : \mathbb{C}^{|\mathbb{N}|} \to \mathbb{C}^{|\mathbb{N}|}$ is defined to characterise indexed algebras:

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

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 indexed algebras are precisely objects in `Ix-Alg`. Since an morphism $(\triangleright A) \to A$ is in bijection with $A \to (\triangleleft A)$, an indexed algebra can be given by the following tuple:

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

the intuition for indexed algebras is that the carrier A_i at level i interprets programs enclosed by i layers of scopes, and thus it must provide a way p to *p*romote the carrier to the next level when it enters a scope, and a way d to *d*emote the carrier when it exists a scoped operation, and additionally the morphism a interprets ordinary *a*lgebraic operations.

Example 2.1. We reformulate our programming example of the indexed algebra for nondeterministic choice with *once* shown in Section 2.1 here. Let $\mathbb{C}$ be the category of sets and $\Sigma X = 1 + X \times X$ representing the coproduct of algebraic operations *fail* and *or*, and $\Gamma X = X$ representing the unary scoped operation *once*. Let $\mathit{List} : \mathtt{Set} \to \mathtt{Set}$ be the endofunctor mapping a set X to the set of lists whose elements are in X. For any set X, we define an 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 structure maps

$$a_i(\iota_1\ \star) = \mathit{nil} \qquad a_i(\iota_2\ \langle x, y\rangle) = x + y \qquad d_i(\mathit{nil}) = \mathit{nil} \qquad d_i(\mathit{cons}\ x\ xs) = x \qquad p_i(x) = \mathit{cons}\ x\ \mathit{nil}$$

where $\star$ is the only element in singleton set 1, and *nil* is the empty list, and $+$ is list concatenation, and *cons x xs* is the list with an element x in front of xs.

The adjunction $\mathtt{Free}_{\mathtt{Ix}} \dashv \mathtt{U}_{\mathtt{Ix}}$ in (2) is the free-forgetful adjunction for endofunctor $\mathtt{Ix}$ on $\mathbb{C}^{|\mathbb{N}|}$. The other adjunction $\uparrow \dashv \downarrow$ is given by

$$(\uparrow X)_0 = X \qquad (\uparrow X)_{i+1} = 0 \qquad \downarrow A = A_0$$

As shown by Piróg et al. [2018], the monad $\downarrow \mathtt{U}_{\mathtt{Ix}}\mathtt{Free}_{\mathtt{Ix}} \uparrow$ induced by adjunction (2) is isomorphic to monad E modelling syntax. Thus we can interpret scoped operations with an indexed algebra by

$$\mathit{eval}_{\langle A,a,d,p\rangle}\ g = \downarrow \mathtt{U}_{\mathtt{Ix}}(\epsilon_{\langle A,a,d,p\rangle} \cdot \mathtt{Free}_{\mathtt{Ix}} \uparrow g) : EA \cong (\downarrow \mathtt{U}_{\mathtt{Ix}}\mathtt{Free}_{\mathtt{Ix}} \uparrow)A \to A_0 \tag{3}$$

where ϵ is the counit of the adjunction (2). The implementation *hfold* in Figure 1 is not a direct implementation of *eval* though. The connection between them will be the subject of Section 6.2.

In summary, the syntax of scoped operations is modelled by an endofunctor $E = \mu G$ (Definition 2.1), which is equipped with the monadic structure of $(\Sigma + \Gamma E)^*$. One way to give semantics to scoped operations is by indexed algebras whose associated adjunction (2) induces a monad isomorphic to E. In the following sections, we will present other kinds of algebras for scoped operations that avoid certain drawbacks of indexed algebras, and for each way of interpreting scoped operations, there is always an adjunction inducing an monad isomorphic to E.

3 INTERPRETING SCOPED OPERATIONS WITH EILENBERG-MOORE ALGEBRAS

A downside of indexed algebras is that their implementation needs types indexed by natural numbers, which do not exist in the majority of programming languages nowadays. In this section, we introduce a more implementation-friendly but less structured approach to scoped operations based on the *Eilenberg-Moore adjunction* (Section 3.1) and we show it implementable in a programming language with only higher-order functions and general recursion (Section 3.2).

3.1 Eilenberg-Moore Algebras of Scoped Effects

For a brief background, given any monad $M : \mathbb{C} \to \mathbb{C}$, an Eilenberg-Moore algebra (*EM* algebra for short) [Mac Lane 1998] is an object X in $\mathbb{C}$ together with a structure map $\alpha : MX \to X$ that "behaves well" w.r.t. the unit and multiplication of the monad as described in the following diagram:

For a pair of Eilenberg-Moore algebras $\langle X, \alpha\rangle$ and $\langle X', \alpha'\rangle$, a morphism between them is a morphism $f : X \to X'$ in $\mathbb{C}$ that respects the algebra structure, that is $f \cdot \alpha = \alpha' \cdot Mf$. The category of Eilenberg-Moore algebras and their morphisms is denoted by $\mathbb{C}^M$. An adjunction $L \dashv R$ for some $L : \mathbb{C} \to \mathbb{D}$ is *monadic* if $\mathbb{D}$ and $\mathbb{C}^{RL}$ are *equivalent*, and is *strictly* monadic if these two categories are *isomorphic*. Notably, adjunction $\mathtt{Free}_F \dashv \mathtt{U}_F$ is strictly monadic, and thus $F\text{-}\mathtt{Alg}$ is isomorphic to $\mathbb{C}^{F^*}$.

Since handlers of algebraic operations are essentially Eilenberg-Moore algebras over the free monad Σ^* of the signature functor Σ, we are interested in using the Eilenberg-Moore algebras over the syntax monad E (Definition 2.1) for interpretation as well. As we explained in Section 2, the monadic structure of E is given by the free monad $(\Sigma + \Gamma E)^*$, so the Eilenberg-Moore algebras of E are equivalent to those of $(\Sigma + \Gamma E)^*$. Furthermore, since the adjunction inducing $(\Sigma + \Gamma E)^*$ is *strictly monadic* [Mac Lane 1998], the Eilenberg-Moore algebras of $(\Sigma + \Gamma E)^*$ are equivalent to the algebras over the *endofunctor* $(\Sigma + \Gamma E)$, which are objects X in $\mathbb{C}$ paired with a morphism $\Sigma X + \Gamma E X \to X$ with no coherence conditions.

Based on these observations, we obtain a way of interpreting scoped operations based on the free-forgetful adjunction $\mathsf{Free}_{\Sigma+\Gamma E} \dashv \mathsf{U}_{\Sigma+\Gamma E}$: give an EM algebra of E in the form of

$$\langle X : \mathbb{C},\ \alpha_\Sigma : \Sigma X \to X,\ \alpha_\Gamma : \Gamma E X \to X \rangle$$

then for any morphism $g : A \to X$ and $A : \mathbb{C}$, the interpretation of EA by g and this EM algebra is

$$\mathit{eval}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g = \mathsf{U}_{\Sigma+\Gamma E}(\epsilon_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle} \cdot \mathsf{Free}_{\Sigma+\Gamma E}\ g) : EA \cong (\Sigma + \Gamma E)^* A \to X \tag{4}$$

The morphism g transforms the returned value A into the carrier X, so it corresponds to the 'return clause' of effect handlers [Plotkin and Pretnar 2013]. The formula (4) can be turned into a more direct form by the following standard result relating free algebras and initial algebras [Barr 1970].

Lemma 3.1. *For any functor $F : \mathbb{C} \to \mathbb{C}$ on some category $\mathbb{C}$, if the initial algebra $\langle \mu Y.\ X + FY, \mathit{in}\rangle$ exists for any $X : \mathbb{C}$, then the free algebra $(F^*X, \mathit{op}_X : FF^*X \to F^*X)$ is isomorphic to*

$$\langle \mu Y.\ X + FY,\ \ \mathit{in} \cdot \iota_2 : F(\mu Y.\ X + FY) \to (\mu Y.\ X + FY)\rangle$$

Moreover, if the isomorphism between them is $\phi : F^*X \to \mu Y.X + FY$, then the unit and counit of the free-forgetful adjunction $\mathsf{Free}_F \dashv \mathsf{U}_F$ are

$$\eta_X = \phi^{-1} \cdot \mathit{in} \cdot \iota_1 : X \to F^*X \qquad \epsilon_{\langle C,\beta:FC\to C\rangle} = (\!|[\mathit{id}, \beta]|\!) \cdot \phi : \langle F^*C, \mathit{op}_C\rangle \to \langle C, \beta\rangle \tag{5}$$

and the multiplication of the monad F^*X satisfies

$$\mu_X = \mathsf{U}_F(\epsilon_{\mathsf{Free}_F X}) = (\!|[\mathit{id}, \mathit{op}_X]|\!) \cdot \phi : F^*(F^*X) \to F^*X$$

Theorem 3.2 (Interpreting with EM Algebras)**.** *Given an Eilenberg-Moore algebra carried by $X : \mathbb{C}$ with structure maps $\alpha_\Sigma : \Sigma X \to X$ and $\alpha_\Gamma : \Gamma E X \to X$, for any morphism $g : A \to X$ in $\mathbb{C}$ for some A, the interpretation of EA with this algebra and g satisfies*

$$\mathit{eval}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g = [g,\ \alpha_\Sigma \cdot \Sigma(\mathit{eval}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g),\ \alpha_\Gamma \cdot \Gamma\Sigma(\mathit{eval}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g)] \cdot \mathit{in}^\circ_A \tag{6}$$

where $\mathit{in}^\circ : E \to Id + \Sigma E + \Gamma E E$ is the isomorphism between E and GE.

Proof. It directly follows from the formula of ϵ (5) for the free-forgetful adjunction. □

3.2 Implementation of EM Algebras of Scoped Effects

The chief advantage of Eilenberg-Moore algebras of scoped operations is its simplicity in terms of implementation. Based on our Haskell representation *Prog* Σ Γ of syntax trees of programs in Section 2.1, an Eilenberg-Moore algebra is a carrier type x with two functions $\Sigma\ x \to x$ and $\Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ x) \to x$ where Σ and Γ are the signature functors of algebraic and scoped operations. We represent these data as a record

$$\textbf{data}\ \mathit{EMAlg}\ \Sigma\ \Gamma\ x = \mathit{EM}\ \{\mathit{call}_{EM} :: \Sigma\ x \to x, \mathit{enter}_{EM} :: \Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ x) \to x\}$$

with two components respectively interpreting *call*s to algebraic operations and *enter*ing scopes. Then formula (6) can be straightforwardly translated into a recursive program interpreting programs with an EM algebra:

$$\begin{aligned}
&\mathit{eval}_{EM} :: (\mathit{Functor}\ \Sigma, \mathit{Functor}\ \Gamma) \Rightarrow (\mathit{EMAlg}\ \Sigma\ \Gamma\ x) \to (a \to x) \to \mathit{Prog}\ \Sigma\ \Gamma\ a \to x\\
&\mathit{eval}_{EM}\ \mathit{alg}\ \mathit{gen}\ (\mathit{Return}\ x) = \mathit{gen}\ x\\
&\mathit{eval}_{EM}\ \mathit{alg}\ \mathit{gen}\ (\mathit{Call}\ \mathit{op}) \ \ = (\mathit{call}_{EM}\ \mathit{alg} \cdot \mathit{fmap}\ (\mathit{eval}_{EM}\ \mathit{alg}\ \mathit{gen}))\ \mathit{op}\\
&\mathit{eval}_{EM}\ \mathit{alg}\ \mathit{gen}\ (\mathit{Enter}\ \mathit{op}) = (\mathit{enter}_{EM}\ \mathit{alg} \cdot \mathit{fmap}\ (\mathit{fmap}\ (\mathit{eval}_{EM}\ \mathit{alg}\ \mathit{gen})))\ \mathit{op}
\end{aligned}$$

Example 3.1. The scoped operation *once* in Section 2 can be interpreted by the following EM algebra:

```
onceAlgEM :: EMAlg Choice Once [a]
onceAlgEM = EM {..} where
  call_EM :: Choice [a] → [a]
  call_EM Fail      = []
  call_EM (Or x y) = x ++ y

enter_EM :: Once (Prog Choice Once [a]) → [a]
enter_EM (Once p) =
  case eval_EM onceAlgEM (λx → [x]) p of
    []    → []
    (x: _) → x
```

Note that this algebra interprets the program inside the scope of *once* by a recursive call to $eval_{EM}$ with the algebra itself.

As demonstrated in the example above, EM algebras have complete control over how to handle the program inside scopes, which can be *considered harmful* since it does not adequately reflect the structures in the algebras of scoped operations, making optimisation and reasoning more difficult. Yet, the merit of EM algebras is their simplicity—their implementation is readily ported to any language with higher-order functions and inductive datatypes, which is not a strong requirement.

4 INTERPRETING SCOPED OPERATIONS WITH FUNCTORIAL ALGEBRAS

We have seen that Eilenberg-Moore algebras are easier to implement than indexed algebras but less structured. In this section, we present another way of interpreting scoped operations by *functorial algebras*, which we believe is at a sweet point in the trade-off between structuredness and simplicity. The idea of functorial algebras stems from the fact that the syntax of scoped operations is modelled by an initial algebra $E = \mu G$ in $\mathbb{C}^{\mathbb{C}}$. Following the fruitful line of research on initial algebra semantics [Goguen et al. 1977; Hagino 1987; Johann and Ghani 2007, 2008], a natural way to interpret E is by an endofunctor H carrying a G-algebra $\alpha^G : GH \to H$, which then induces the catamorphism $(\!|\alpha^G|\!) : E \to H$ in $\mathbb{C}^{\mathbb{C}}$. However, since we presuppose that programs live in the base category $\mathbb{C}$, at the end of the day, we need to interpret programs by morphisms $EA \to X$ for some A and X in the base category $\mathbb{C}$. Hence we additionally equip a G-algebra H with an object $X : \mathbb{C}$ that interprets the part of a program not inside any scoped operation, and use the endofunctor H only for interpreting the inner layers of the program. We call such a pair H and X a *functorial algebra*.

After defining functorial algebras formally (Section 4.1), we show that there is an adjunction between the category of functorial algebras and the base category (Section 4.2), and importantly, this adjunction induces a monad isomorphic to E that models the syntax of programs with scoped operations. Hence functorial algebras can be used to interpret scoped operations modelled by the monad E. Finally, we show a Haskell implementation of interpreting scoped operations of functorial algebras and examples (Section 4.3).

4.1 Functorial Algebras

A functorial algebra is carried by a pair of an endofunctor $H : \mathbb{C} \to \mathbb{C}$ and an object X in $\mathbb{C}$. The endofunctor H is equipped with a morphism α^G from GH (i.e. $Id + \Sigma H + \Gamma HH$) to H in $\mathbb{C}^{\mathbb{C}}$, and the object X is equipped with a morphism $\alpha^I : \Sigma X + \Gamma HX \to H$ in $\mathbb{C}$. The intuition is that given a program of type $EX \cong X + \Sigma EX + \Gamma EEX$, the middle E in ΓEE corresponds to the part of the program inside scoped operations, and it is interpreted as H by α^G. After this, α^I interprets the outermost layer of the program as X just as with interpreting free monads with no scoped operations.

We formalise the idea on the product category $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$. Let $I : \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}$ be a (bi-)functor such that $I_H X = \Sigma X + \Gamma HX$ for any $H : \mathbb{C}^{\mathbb{C}}$ and $X : \mathbb{C}$. Its action on morphisms is given by $I_\sigma f = \Sigma f + \Gamma(\sigma \circ f)$ for any $\sigma : H \to H'$ and $f : X \to X'$ where $\circ$ is horizontal composition.

Then we define an endofunctor Fn on $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ by $\mathsf{Fn}\langle H, X\rangle = \langle GH, I_H X\rangle$ with the evident action on morphisms.

Definition 4.1 (Functorial Algebras)**.** A *functorial algebra* is a Fn-algebra $\langle H, X\rangle$ in $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ paired with a structure map $\mathsf{Fn}\langle H, X\rangle \to \langle H, X\rangle$, or equivalently, a functorial algebra is a tuple

$$\left\langle \langle H : \mathbb{C}^{\mathbb{C}},\ X : \mathbb{C}\rangle,\ \langle \alpha^G : GH \to H,\ \alpha^I : I_H X \to X\rangle \right\rangle$$

Example 4.1. We reformulate the indexed algebra of nondeterminism with *once* here. Assuming the notation in Example 2.1, we define natural transformations $\alpha^{\Sigma} : \Sigma \mathit{List} \to \mathit{List}$ and $\alpha^{\Gamma} : \Gamma \mathit{List}\mathit{List} \to \mathit{List}$ by

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

Then for any set X, $\langle \mathit{List}, \mathit{List}\ X\rangle$ carries a functorial algebra with structure maps

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

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

Example 4.2. The last example demonstrates a common pattern in practice that the object component X of a functorial algebra is simply HX where H is the endofunctor component, but it is not necessarily so. For example, if one is only interested in the final number of possible outcomes, then a functorial algebra is $\langle \mathit{List}, \mathbb{N}, \alpha^G, \alpha^I\rangle$ where

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

4.2 An Adjunction for Functorial Algebras

In this subsection we show how functorial algebras can be used to interpret programs involving scoped operations modelled by the monad E. We first construct an adjunction $\Uparrow \dashv \Downarrow$ (7) between the base category $\mathbb{C}$ and $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$, which is then composed with the free-forgetful adjunction $\mathsf{Free}_{\mathsf{Fn}} \dashv \mathsf{U}_{\mathsf{Fn}}$ between $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ and $\mathsf{Fn}\text{-}\mathsf{Alg}$. The resulting adjunction (8) is proven to induce a monad isomorphic to E (Theorem 4.4), and by the adjoint-theoretic approach to syntax and semantics (Section 2.2.2), this adjunction provides a means to interpret scoped operations modelled with the monad E.

We start with constructing $\Uparrow \dashv \Downarrow$ between $\mathbb{C}^{\mathbb{C}}$ and $\mathbb{C}$. Let $\Uparrow : \mathbb{C} \to \mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ be the functor mapping an object X to $\langle \underline{0}, X\rangle$ where $\underline{0} : \mathbb{C}^{\mathbb{C}}$ is the constant functor to the initial object in $\mathbb{C}$. The action of $\Uparrow$ on morphisms is $\Uparrow f = \langle !, f\rangle$, where $!$ is the unique morphism from the initial object. The functor $\Uparrow$ is left adjoint to the projection functor, which we call $\Downarrow : \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}$, mapping $\langle H, X\rangle$ to X with the obvious action on morphisms.

Lemma 4.1. *For all H in $\mathbb{C}^{\mathbb{C}}$ and X in $\mathbb{C}$, there is an isomorphism of of hom-sets*

$$\lfloor \cdot \rfloor : \mathbb{C}^{\mathbb{C}} \times \mathbb{C}(\Uparrow A, \langle H, X\rangle) \cong \mathbb{C}(A, \Downarrow \langle H, X\rangle) : \lceil \cdot \rceil \tag{7}$$

natural in A and $\langle H, X\rangle$.

Proof. Given a map $f : \Uparrow A \to \langle H, X\rangle$, that is a map $\langle f_1, f_2\rangle : \langle \underline{0}, A\rangle \to \langle H, X\rangle$, we define $\lceil f \rceil = f_2$. Conversely, given a map $g : A \to \Downarrow \langle H, X\rangle$, that is $g : A \to X$, we define $\lfloor g \rfloor = \langle !, g\rangle$ where $! : \underline{0} \to H$ is the unique map from the initial object $\underline{0}$ in $\mathbb{C}^{\mathbb{C}}$. Then it is easy to see that $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ are mutual inverses, and that this isomorphism satisfies the naturality requirements. □

Assuming enough initial algebras exist, let $\mathsf{Free}_{\mathsf{Fn}}$ be the functor mapping an object $\langle H, X\rangle$ in $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ to the free Fn-algebra and U_{Fn} be the forgetful functor. Then we have two adjunctions

depicted in the following diagram:

$$\mathtt{Fn\text{-}Alg} \underset{\mathtt{U_{Fn}}}{\overset{\mathtt{Free_{Fn}}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \underset{\Downarrow}{\overset{\Uparrow}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C} \circlearrowleft T \tag{8}$$

The two adjunctions can be composed to an adjunction $\mathtt{Free_{Fn}} \Uparrow \dashv \Downarrow \mathtt{U_{Fn}}$ between $\mathtt{Fn\text{-}Alg}$ and $\mathbb{C}$. We call the monad induced by this adjunction $T = \Downarrow \mathtt{U_{Fn}} \mathtt{Free_{Fn}} \Uparrow$.

In the rest of this section, we prove that T is isomorphic to E in the category of monads, which plays an essential role in this paper, since it allows us to interpret scoped operations E with functorial algebras. We first establish a lemma characterising the free $\mathtt{Fn}$-algebra on the product category $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ in terms of the free algebras in $\mathbb{C}$ and $\mathbb{C}^{\mathbb{C}}$. The intuition is that the first component of $\mathtt{Fn}\langle H, X\rangle$ is GH, which does not depend on X. Thus the first component of $\mathtt{Free_{Fn}}\langle H, X\rangle$ should be determined only by H too.

Lemma 4.2. *When G^*H and $(I_{G^*H})^*X$ exists for any $\langle H, X\rangle : \mathbb{C}^{\mathbb{C}} \times C$, there is a natural isomorphism between $\mathtt{Free_{Fn}}$ and the functor $\mathtt{FFn} : \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathtt{Fn\text{-}Alg}$ such that*

$$\langle H, X\rangle \mapsto \left\langle \langle G^*H, (I_{G^*H})^*X\rangle, \langle op_H^{G^*}, op_X^{(I_{G^*H})^*}\rangle \right\rangle$$

with the evident action on morphisms.

Proof. By Lemma 3.1, the free $\mathtt{Fn}$-algebra generated by $\langle H, X\rangle$ can be constructed from the initial algebra of a functor $\mathtt{Fn}_{\langle H,X\rangle} : \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ where $\mathtt{Fn}_{\langle H,X\rangle} Y = \langle H, X\rangle + \mathtt{Fn} Y$. Then we show that an initial $\mathtt{Fn}_{\langle H,X\rangle}$-algebra carried by $\langle G^*H, (I_{G^*H})^*X\rangle$ with structure map

$$\langle i_1, i_2\rangle : \mathtt{Fn}_{\langle H,X\rangle}\langle G^*H, (I_{G^*H})^*X\rangle = \langle H + GH, X + I_{G^*H}((I_{G^*H})^*X)\rangle \to \langle G^*H, (I_{G^*H})^*X\rangle$$

where $i_1 = [\eta_H^{G^*}, \mathrm{op}_H^{G^*}] : H + G(G^*H) \to G^*H$ and

$$i_2 = [\eta_X^{(I_{G^*H})^*}, \mathrm{op}_X^{(I_{G^*H})^*}] : X + I_{G^*H}((I_{G^*H})^*X) \to (I_{G^*H})^*X.$$

To see that this $\mathtt{Fn}_{\langle H,X\rangle}$-algebra is initial, consider any $\langle C, D\rangle$ in $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ with structure map $\langle j_1, j_2\rangle : \mathtt{Fn}_{\langle H,X\rangle}\langle C, D\rangle \to \langle C, D\rangle$. We have

$$\begin{aligned}
&\langle k_1, k_2\rangle : \mathtt{Fn}_{\langle H,X\rangle}\text{-}\mathtt{Alg}\left(\left\langle \langle G^*H, (I_{G^*H})^*X\rangle,\ \langle i_1, i_2\rangle \right\rangle,\ \left\langle \langle C, D\rangle, \langle j_1, j_2\rangle \right\rangle\right)\\
\Leftrightarrow &\left(k_1 \in (H + G-)\text{-}\mathtt{Alg}\left(\langle G^*H, i_1\rangle,\ \langle C, j_1\rangle\right)\right)\\
&\qquad \wedge \left(k_2 \in (X + I_{G^*H}-)\text{-}\mathtt{Alg}\left(\langle I_{G^*H}X, i_2\rangle,\ \langle D, j_2 \cdot (X + I_{k_1}\mathrm{id})\rangle\right)\right)\\
\Leftrightarrow\ & k_1 = \lceil j_1 \cdot \iota_1 \rceil_{\langle C, j_1 \cdot \iota_2\rangle} \ \wedge\ k_2 = \lceil j_2 \cdot (X + I_{k_1}) \cdot \iota_1 \rceil_{\langle D, j_2 \cdot (X + I_{k_1}) \cdot \iota_2\rangle}
\end{aligned}$$

where we use subscripts of $\lceil \cdot \rceil$ to indicate the B for some $\lceil f \rceil : LA \to B$. The calculation shows that the $\mathtt{Fn}_{\langle H,X\rangle}$-algebra homomorphism (k_1, k_2) uniquely exists, and thus $\langle G^*H, (I_{G^*H})^*X\rangle$ with structure map $\langle i_1, i_2\rangle$ is initial. Then by Lemma 3.1, this initial algebra gives the free $\mathtt{Fn}$-algebra generated by $\langle H, X\rangle$, and thus we have the isomorphism between $\mathtt{Free_{Fn}}$ and $\mathtt{FFn}$ in the lemma. □

This characterisation of free $\mathtt{Fn}$-algebras also allows us to express the unit and counit of the adjunction $\mathtt{Free_{Fn}} \dashv \mathtt{U_{Fn}}$ in terms of those of some simpler adjunctions.

Lemma 4.3. *Letting the ϕ be the isomorphism in Lemma 4.2, the unit of adjunction $\mathtt{Free_{Fn}} \dashv \mathtt{U_{Fn}}$ is*

$$\eta_{\langle H,X\rangle} = \langle H, X\rangle \xrightarrow{\langle \eta_H^{G^*}, \eta_X^{(I_{G^*H})^*}\rangle} \langle G^*H, (I_{G^*H})^*X\rangle \xrightarrow{\phi^{-1}} \mathtt{Fn}^*\langle H, X\rangle$$

and its counit ϵ at some Fn*-algebra* $\langle\langle H, X\rangle, \langle\beta_1, \beta_2\rangle\rangle$ *is*

$$\epsilon = \mathsf{Free}_{\mathsf{Fn}}\langle H, X\rangle \xrightarrow{\phi} \left\langle\langle G^*H, (I_{G^*}H)^*X\rangle, \langle op_H^{G^*}, op_X^{(I_{G^*}H)^*}\rangle\right\rangle \xrightarrow{\langle e_1, e_2\rangle} \langle\langle H, X\rangle, \langle\beta_1, \beta_2\rangle\rangle$$

where $e_1 = \mathsf{U}_G(\epsilon^{G^*}_{\langle H, \beta_1\rangle})$ *and* $e_2 = \mathsf{U}_{I_{G^*}H}(\epsilon^{(I_{G^*}H)^*}_{\langle X, \beta_2 \cdot I_{e_1}\rangle})$.

Proof Sketch. It can be calculated from (5) and Lemma 4.2. □

Theorem 4.4. *The monad E is isomorphic to T in the category of monads.*

Proof. Since $E \cong (\Sigma + \Gamma E)^* = (I_E)^*$ as monads, it is sufficient to show that T is isomorphic to $(I_E)^*$ as monads. Recall that $E = \mu G \cong (G^*\underline{0})$ as endofunctors. Let $\psi : G^*\underline{0} \to E$ be the isomorphism. Then by Lemma 4.2, for any $X : \mathbb{C}$,

$$TX = (\Downarrow \mathsf{U}_{\mathsf{Fn}} F_{\mathsf{Fn}} \Uparrow)X = (\Downarrow \mathsf{Fn}^*)\langle\underline{0}, X\rangle \overset{\Downarrow\phi}{\cong} \Downarrow(G^*\underline{0}, \langle I_{G^*}\underline{0}\rangle^* X) = (I_{G^*}\underline{0})^* X \overset{(I_\psi)^*}{\cong} (I_E)^* X \tag{9}$$

where ϕ is the isomorphism between Fn^* and $\mathsf{U}_{\mathsf{Fn}}\mathsf{FFn}$ as in Lemma 4.2. Thus T is isomorphic to $(I_E)^*$ as endofunctors.

What remains is to show that the isomorphism (9) preserves their units and multiplications. The unit of T is precisely the unit of the adjunction $(\mathsf{Free}_{\mathsf{Fn}} \Uparrow) \dashv (\Downarrow \mathsf{U}_{\mathsf{Fn}})$ composed from the adjunctions $\Uparrow \dashv \Downarrow$ and $\mathsf{Free}_{\mathsf{Fn}} \dashv \mathsf{U}_{\mathsf{Fn}}$. Therefore the unit of T is $\eta^T_X = \Downarrow(\eta^{\mathsf{Fn}^*}_{\Uparrow X}) \cdot \eta^{\Downarrow\Uparrow}_X$ where $\eta^{\Downarrow\Uparrow}$ and η^{Fn^*} are the units of the adjunctions $\Uparrow \dashv \Downarrow$ and $\mathsf{Free}_{\mathsf{Fn}} \dashv \mathsf{U}_{\mathsf{Fn}}$ respectively. Hence by Lemma 4.3, we have

$$\eta^T_X = \Downarrow(\eta^{\mathsf{Fn}^*}_{\Uparrow X}) \cdot \eta^{\Downarrow\Uparrow}_X = \Downarrow(\eta^{\mathsf{Fn}^*}_{\Uparrow X}) \cdot \mathsf{id} = \Downarrow(\phi^{-1} \cdot \langle \eta^{G^*}_{\underline{0}}, \eta^{(I_{G^*}\underline{0})^*}_X\rangle) = \Downarrow(\phi^{-1}) \cdot \eta^{(I_{G^*}\underline{0})^*}_X = \Downarrow\phi \cdot (I_\psi{}^*)^{-1} \cdot \eta^{(I_E)^*}_X$$

which shows that the isomorphism (9) preserves the units of T and $(I_E)^*$.

Proving the preservation of the multiplications of the two monads is also direct verification but slightly more involved. By definition, $\mu^T_X = \Downarrow \mathsf{U}_{\mathsf{Fn}} \epsilon^T_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X}$ where ϵ^T is the counit of the adjunction $(\mathsf{Free}_{\mathsf{Fn}} \Uparrow) \dashv (\Downarrow \mathsf{U}_{\mathsf{Fn}})$ satisfying

$$\epsilon^T_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X} = \epsilon^{\mathsf{Fn}^*}_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X} \cdot \mathsf{Free}_{\mathsf{Fn}}(\epsilon^{\Downarrow\Uparrow}_{\mathsf{Fn}^* \Uparrow X}) = \epsilon^{\mathsf{Fn}^*}_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X} \cdot \mathsf{Free}_{\mathsf{Fn}}\langle !, \mathsf{id}\rangle$$

where ! is the unique G-algebra homomorphism from $G^*\underline{0}$ to $G^*G^*\underline{0}$. Then by Lemma 4.3, we have $\epsilon^{\mathsf{Fn}^*}_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X} = \langle e_1, e_2\rangle \cdot \phi$ where $e_1 = \mathsf{U}_G(\epsilon^{G^*}_{(G^*\underline{0}, \mathsf{op}^{G^*}_{\underline{0}})}) : G^*G^*\underline{0} \to G^*\underline{0}$ and $e_2 = \mathsf{U}_{I_{G^*G^*}\underline{0}}(\epsilon_b^{(I_{G^*G^*}\underline{0})^*})$ where b is the $I_{G^*G^*}\underline{0}$-algebra $\langle (I_{G^*}\underline{0})^* X, \mathsf{op}_X^{(I_{G^*}\underline{0})^*} \cdot I_{e_1}\rangle$. Hence we have

$$\begin{aligned}
\mu^T_X &= \Downarrow \mathsf{U}_{\mathsf{Fn}}(\epsilon^T_{\mathsf{Free}_{\mathsf{Fn}} \Uparrow X}) \\
&= \mathsf{U}_{I_{G^*G^*}\underline{0}}(\epsilon_b^{(I_{G^*G^*}\underline{0})^*}) \cdot \Downarrow\phi \cdot \Downarrow \mathsf{U}_{\mathsf{Fn}} \mathsf{Free}_{\mathsf{Fn}}(!, \mathsf{id}) \\
&= \mathsf{U}_{I_{G^*G^*}\underline{0}}(\epsilon_b^{(I_{G^*G^*}\underline{0})^*}) \cdot \Downarrow\phi \cdot \Downarrow(\phi^{-1} \cdot \langle G^*!, (I_{G^*}!)^*\rangle \cdot \phi) \\
&= \mathsf{U}_{I_{G^*G^*}\underline{0}}(\epsilon_b^{(I_{G^*G^*}\underline{0})^*}) \cdot (I_{G^*}!)^* \cdot \Downarrow\phi
\end{aligned}$$

where $(I_{G^*}!)^*$ is a natural transformation from $(I_{G^*}\underline{0})^*$ to $(I_{G^*G^*}\underline{0})^*$. Then by *base functor fusion* [Hinze 2013] i.e. the naturality of the free-forgetful adjunction in the base functor, we have

$$\mu^T_X = \mathsf{U}_{I_{G^*G^*}\underline{0}}(\epsilon_b^{(I_{G^*G^*}\underline{0})^*}) \cdot (I_{G^*}!)^* \cdot \Downarrow\phi = \mathsf{U}_{I_{G^*}\underline{0}}(\epsilon_{b'}^{(I_{G^*}\underline{0})^*}) \cdot \Downarrow\phi$$

where b' is the $I_{G^*}\underline{0}$-algebra with the same carrier as b and with b's structure map precomposed with $I_{G^*}!$:

$$\mathsf{op}_X^{(I_{G^*}\underline{0})^*} \cdot I_{e_1} \cdot I_{G^*}! = \mathsf{op}_X^{(I_{G^*}\underline{0})^*} \cdot I_{e_1 \cdot !} = \mathsf{op}_X^{(I_{G^*}\underline{0})^*} \cdot I_{\mathsf{id}} = \mathsf{op}_X^{(I_{G^*}\underline{0})^*}$$

where the second equality follows from that $e_1 \cdot ! : G^*\underline{0} \to G^*\underline{0}$ is a G-algebra homomorphism, and thus $e_1 \cdot ! = \mathrm{id}$ since $G^*\underline{0}$ is an initial G-algebra. Finally we have

$$\mu_X^T = \mathsf{U}_{I_{G^*\underline{0}}}(\epsilon^{(I_{G^*\underline{0}})^*}_{\langle (I_{G^*\underline{0}})^* X, \mathrm{op}_X^{(I_{G^*\underline{0}})^*}\rangle}) \cdot \Downarrow\phi = \mathsf{U}_{I_E}(\epsilon^{I_E^*}_{\langle I_E^* X, \mathrm{op}_X^{(I_E)^*}\rangle}) \cdot ((I_\psi)^* \cdot \Downarrow\phi) \cdot T((I_\psi)^* \cdot \Downarrow\phi)$$

which means exactly that the isomorphism (9) preserves the multiplications of T and $(I_E)^*$. □

Remark 4.1. In general, the adjunction $\mathtt{Free}_{\mathsf{Fn}} \Uparrow \dashv \Downarrow \mathsf{U}_{\mathsf{Fn}}$ is *not* monadic since the right adjoint $\Downarrow \mathsf{U}_{\mathsf{Fn}}$ does not reflect isomorphisms, which is a necessary condition for it to be monadic by Beck's monadicity theorem [Mac Lane 1998]. This entails that the category `Fn-Alg` of functorial algebras is not equivalent to the category of Eilenberg-Moore algebras. Nonetheless, as we will see later in Section 5, functorial algebras and Eilenberg-Moore algebras have the same expressive power for interpreting scoped operations in the base category.

4.3 Implementation

Now our hard work in the previous subsection pays off: for any functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$ as in Definition 4.1, and any morphism $g : A \to X$ in the base category $\mathbb{C}$, there is a morphism

$$\mathit{eval}_{\langle H,X,\alpha^G,\alpha^I\rangle}\ g = \Downarrow \mathsf{U}_{\mathsf{Fn}}(\epsilon_{\langle H,X,\alpha^G,\alpha^I\rangle} \cdot \mathtt{Free}_{\mathsf{Fn}} \Uparrow g) : TA \cong EA \to X \tag{10}$$

which interprets programs with scoped operations modelled as EA by the functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$. Furthermore, we can derive a recursive formula for this interpretation morphism using Lemma 4.3, which can be straightforwardly translated into an implementation.

Lemma 4.5 (Interpreting with Functorial Algebras)**.** *For any functorial algebra $F = \langle H, X, \alpha^G, \alpha^I\rangle$ as in Definition 4.1, and any morphism $g : A \to X$ for some A in the base category $\mathbb{C}$, letting $h = (\!|\alpha^G|\!) : E \to H$ be the catamorphism from the initial G-algebra E to $\langle H, \alpha^G\rangle$, the interpretation of EA with this algebra F and g satisfies*

$$\mathit{eval}_F\ g = [g,\ \alpha^I_\Sigma \cdot \Sigma(\mathit{eval}_F\ g),\ \alpha^I_\Gamma \cdot \Gamma h_X \cdot \Gamma E(\mathit{eval}_F\ g)] \cdot \mathit{in}^\circ_A\ :\ EA \to X \tag{11}$$

where $\mathit{in}^\circ : E \to Id + \Sigma E + \Gamma EE$ is the isomorphism between E and GE, and $\alpha^I_\Sigma = \alpha^I \cdot \iota_1 : \Sigma X \to X$, and $\alpha^I_\Gamma = \alpha^I \cdot \iota_2 : \Gamma HX \to X$ are the two components of $\alpha^I : \Sigma X + \Gamma HX \to X$.

Proof Sketch. It can be calculated by plugging in the formula for ϵ in Lemma 4.3 in (10). □

The formula (11) is readily translated into the Haskell implementation of Figure 2, and a OCaml implementation in a similar spirit is listed and explained in the Appendix. The datatype *EndoAlg* represents α^G; datatype *BaseAlg* corresponds to α^I; function *hcata* implements $(\!|\alpha^G|\!)$; and the three arguments of *eval* implement the three components in (11) respectively.

From a pragmatic perspective, functorial algebras are a sweet spot in the design space because they provide a framework that make it relatively easy to define new algebras. There is no need to resort to the complexities of the indexed types we saw in Section 2, and the algebras are nevertheless just as expressive.

For instance, here is how *list* is defined, using return_{list} and call_{list} from Section 2:

```
list :: Prog Choice Once a → [a]
list = eval (EndoAlg return_list call_list enter_list)
            (BaseAlg call_list enter_list) return_list

enter_list :: Once [[a]] → [a]
enter_list (Once [])       = []
enter_list (Once (xs : xss)) = xs
```

The counterparts to demote_{once} and $\mathit{promote}_{once}$ from the indexed algebra version are enter_{list} and return_{list}. The versions that use functorial algebras do not require any indexed types.

```
data BaseAlg Σ Γ c d =
  BaseAlg { callB  :: Σ d → d
          , enterB :: Γ (c d) → d }

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

hcata :: (Functor Σ, Functor Γ) ⇒ (EndoAlg Σ Γ c) → Prog Σ Γ a → c 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

eval :: (Functor Σ, Functor Γ) ⇒ (EndoAlg Σ Γ x) → (BaseAlg Σ Γ x b) → (a → b) → Prog Σ Γ a → b
eval ealg balg gen (Return x)     = gen x
eval ealg balg gen (Call op)      = (callB balg · fmap (eval ealg balg gen)) op
eval ealg balg gen (Enter scope) = (enterB balg · fmap (hcata ealg · fmap (eval ealg balg gen))) scope
```

Fig. 2. A Haskell implementation of the evaluation function of functorial algebras

Notice that in this handler $call_{list}$ and $enter_{list}$ are reused in the definition of both the base algebra and the endo algebra. This is because the intermediate datatype between a scoped operation and its outer parent is always the same in this instance. This need not be the case in general.

A more complex heuristic than *once* is to consider how depth-bounded search might be implemented. This heuristic allows a program to be annotated with a depth that bounds the branches of nondeterminism by some natural number. As usual, the starting point is to define the syntax of this scoped operation, where *depth d p* will limit the nondeterminsm in the program *p* by the depth *d*:

```
data Depth a = Depth Int a deriving Functor

depth :: Int → Prog Choice Depth a → Prog Choice Depth a
depth d p = Enter (Depth d (fmap return p))
```

Interpreting this syntax requires a carrier that is sensitive to the current depth, so this will be a function from the depth to a list of results:

```
newtype DepthCarrier a = DepthCarrier (Int → [a])
```

As for the handler for programs that have this operation, here is *dbs*:

```
dbs :: Prog Choice Depth a → [a]
dbs = eval (EndoAlg return_dbs call_dbs enter_dbs) (BaseAlg call_list exit_dbs) return_list where
  return_dbs :: a → DepthCarrier a
  return_dbs x = DepthCarrier (const [x])

  call_dbs :: Choice (DepthCarrier a) → DepthCarrier a
  call_dbs Fail = DepthCarrier (const [])
  call_dbs (Or (DepthCarrier fxs) (DepthCarrier fys))
    = DepthCarrier (λd → if d ≡ 0 then [] else fxs (d − 1) ++ fys (d − 1))

  enter_dbs :: Depth (DepthCarrier (DepthCarrier a)) → DepthCarrier a
  enter_dbs (Depth d (DepthCarrier fxs)) =
    DepthCarrier (λd' → concat [fys d' | DepthCarrier fys ← fxs d])

  exit_dbs :: Depth (DepthCarrier [a]) → [a]
  exit_dbs (Depth d (DepthCarrier fxs)) = concat (fxs d)
```

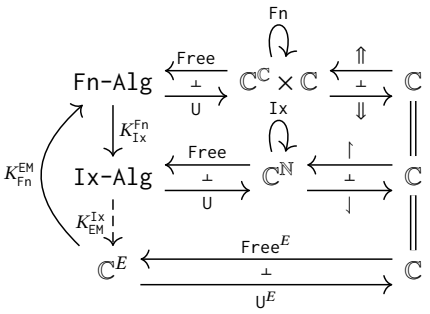


Fig. 3. The resolutions of functorial, indexed and Eilenberg-Moore algebras

Although this handler makes use of the *DepthCarrier a* type to manage the state of the depth counter, this type is not exposed externally. This is because $exit_{dbs}$ converts from the types *DepthCarrier* [*a*] to [*a*], thus hiding the complexities of this semantics from users.

5 COMPARING THE EXPRESSIVITY OF THE MODELS

At this point we have seen three ways for interpreting scoped operations: indexed, Eilenberg-Moore and functorial algebras. They structure interpretation in different ways, making them suitable for different applications. In this section, we compare the expressivity of the three models and show that their expressive power is in fact equivalent. To do this, we construct *comparison functors* between the respective categories of the three kinds of algebras, which translate one kind of algebras to another and *preserve the induced interpretation* in the base category. Categorically, these functors are morphisms in the category of *resolutions* of the monad E (Section 5.1). In this category, the Eilenberg-Moore adjunction is the terminal object, and thus it automatically gives us comparison functors translating indexed and functorial algebras to EM algebras (Section 5.2). To complete the circle of translations, we then construct comparison functors $K^{\mathsf{EM}}_{\mathsf{Fn}} : \mathbb{C}^E \to \mathtt{Fn\text{-}Alg}$ translating EM algebras to functorial ones (Section 5.3) and $K^{\mathsf{Fn}}_{\mathsf{Ix}} : \mathtt{Fn\text{-}Alg} \to \mathtt{Ix\text{-}Alg}$ translating functorial algebras to indexed ones (Section 5.4). The situation is pictured in Figure 3.

5.1 Comparison of Resolutions

We have followed the adjoint approach to syntax and semantics of scoped operations—any adjunction inducing the monad E modelling the syntax of scoped operations provides a way to give semantics to scoped operations. As shown in [Lambek and Scott 1986; Riehl 2017], these adjunctions form a category, on which we compare the three kinds of algebras in the rest of this section.

Definition 5.1 (Resolutions and Comparison Functors)**.** Given a monad $\langle M, \eta, \mu\rangle$ on $\mathbb{C}$, the category $\mathtt{Res}(M)$ of *resolutions* of M has as objects adjunctions

$$\langle\ \mathbb{D} \underset{R}{\overset{L}{\leftrightarrows}} \mathbb{C}\ , \eta : Id \to RL, \epsilon : LR \to Id\rangle$$

whose induced monad RL is M. A morphism from a resolution $\langle \mathbb{D}, L, R, \eta, \epsilon\rangle$ to $\langle \mathbb{D}', L', R', \eta', \epsilon'\rangle$ is a functor $K : \mathbb{D} \to \mathbb{D}'$ such that it commutes with the left and right adjoints

$$KL = L' \qquad \text{and} \qquad R'K = R$$

which is called a *comparison functor*.

As we have seen in previous sections, there are three adjunctions

$$\mathsf{Free}_{\mathtt{Ix}} \restriction \dashv \lfloor \mathsf{U}_{\mathtt{Ix}} \qquad \mathsf{Free}_{\Sigma+\Gamma E} \dashv \mathsf{U}_{\Sigma+\Gamma E} \qquad \mathsf{Free}_{\mathtt{Fn}} \Uparrow \dashv \Downarrow \mathsf{U}_{\mathtt{Fn}}$$

for indexed algebras, EM algebras and functorial algebras respectively, each inducing the monad E up to isomorphism, so they can all be identified with some objects in the category $\mathtt{Res}(E)$. For each resolution $\langle \mathbb{D}, L, R, \eta, \epsilon \rangle$ of E, we use the objects D in $\mathbb{D}$ to interpret scoped operations modelled by E: for any morphism $g : A \longrightarrow RD$ in $\mathbb{C}$, the interpretation of EA by D and g is

$$eval_D\ g = R(\epsilon_D \cdot Lg) : EA = RLA \longrightarrow RD$$

Crucially, we show that interpretations are preserved by comparison functors.

Lemma 5.1 (Preservation of Interpretation)**.** *Let $K : \mathbb{D} \longrightarrow \mathbb{D}'$ 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 $M : \mathbb{C} \longrightarrow \mathbb{C}$. For any object D in $\mathbb{D}$ and any morphism $g : A \longrightarrow RD$ in $\mathbb{C}$, it holds that*

$$eval_D\ g = eval_{KD}\ g : MA \longrightarrow RD(= R'KD) \tag{12}$$

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

Proof. Because $L \dashv R$ and $L' \dashv R'$ induce the same monad, their unit must coincide $\eta = \eta'$. Together with the commutativity properties $KL = L'$ and $R'K = R$, it makes a comparison functor a special case of a *map of adjunctions* [Mac Lane 1998]. Then by Proposition 1 in [Mac Lane 1998, page 99], it holds that $K\epsilon = \epsilon' K$, and we have

$$eval_{KD}\ g = R'(\epsilon'_{KD} \cdot L'g) = R'(K\epsilon_D \cdot L'g) = R\epsilon_D \cdot R'L'g = R\epsilon_D \cdot RLg = eval_D\ g$$

which completes the proof. □

The implication of this lemma is that if there is a comparison functor K from some resolution $L \dashv R$ to $L' \dashv R'$ of the monad E where $L : \mathbb{C} \longrightarrow \mathbb{D}$ and $L' : \mathbb{C} \longrightarrow \mathbb{D}'$, 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 E is not greater than $\mathbb{D}'$, in the sense that every $eval_D\ g$ that one can obtain from $D : \mathbb{D}$ can also be obtained by some algebra in $\mathbb{D}'$, namely, KD. Thus the three kinds of algebras for interpreting scoped operations have the same expressive power if we can construct a circle of comparison functors between their categories, which is what we do in the rest of this section.

5.2 Translating to EM Algebras

As shown in [Mac Lane 1998], 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 *any* resolution to the Eilenberg-Moore resolution. Specifically, given a resolution $\langle \mathbb{D}, L, R \rangle$ of a monad M, the unique comparison functor K from $\mathbb{D}$ to the category $\mathbb{C}^M$ of the Eilenberg-Moore algebras is

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

This result immediately give us comparison functors to Eilenberg-Moore algebras.

Lemma 5.2. *There uniquely exist comparison functors $K^{\mathtt{Ix}}_{\mathtt{EM}} : \mathtt{Ix\text{-}Alg} \longrightarrow \mathbb{C}^E$ and $K^{\mathtt{Fn}}_{\mathtt{EM}} : \mathtt{Fn\text{-}Alg} \longrightarrow \mathbb{C}^E$ from the respective resolutions of indexed algebras and functorial algebras to the resolution of Eilenberg-Moore algebras.*

Specifically, given an indexed algebra $\langle A : \mathbb{C}^{|\mathbb{N}|}, \alpha : \mathtt{Ix}A \longrightarrow A \rangle$ the comparison functor $K^{\mathtt{Ix}}_{\mathtt{EM}}$ translates it into an EM algebra carried by A_0 with structure map $\lfloor \mathsf{U}_{\mathtt{Ix}} \epsilon_{\langle A,\alpha \rangle} : EA_0 \longrightarrow A_0$. As we

mentioned in Section 3.1, an EM algebra of E is equivalent to an algebra of the endofunctor $(\Sigma + \Gamma E)$. In this case, $K^{\mathtt{Ix}}_{\mathtt{EM}}(\langle A, \alpha\rangle)$ is equivalent to the $(\Sigma + \Gamma E)$ algebra carried by $A_0 : \mathbb{C}$ with structure map

$$(\Sigma + \Gamma E)A_0 \xrightarrow{(\Sigma+\Gamma E)(\eta_{A_0})} (\Sigma + \Gamma E)(\Sigma + \Gamma E)^* A_0 \xrightarrow{\mathsf{op}} (\Sigma + \Gamma E)^* A_0 \xrightarrow{\psi} EA_0 \xrightarrow{\downarrow \mathsf{U}_{\mathtt{Ix}} \epsilon_{\langle A,\alpha\rangle}} A_0$$

where ψ is the isomorphism between E and $(\Sigma + \Gamma E)^*$.

5.3 Translating EM Algebras to Functorial Algebras

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

Since $\mathbb{C}^E$ is isomorphic to $(\Sigma + \Gamma E)\text{-}\mathtt{Alg}$, we can define $K^{\mathtt{EM}}_{\mathtt{Fn}}$ on $(\Sigma + \Gamma E)$-algebras instead. Given any $\langle X : \mathbb{C}, \alpha : (\Sigma + \Gamma E)X \to X\rangle$ with no coherence conditions on α_Σ and α_Γ, it is mapped by $K^{\mathtt{EM}}_{\mathtt{Fn}}$ to the functorial algebra

$$\langle E,\ X,\ \mathsf{in} : GE \to E,\ \alpha : (\Sigma + \Gamma E)X \to X\rangle$$

and for any morphism f in $(\Sigma + \Gamma E)\text{-}\mathtt{Alg}$, it is mapped to $\langle \mathsf{id}_E, f\rangle$.

Lemma 5.3. *Functor $K^{\mathtt{EM}}_{\mathtt{Fn}}$ is a comparison functor from the Eilenberg-Moore resolution of E to the resolution $\mathtt{Free}_{\mathtt{Fn}} \Uparrow \dashv \Downarrow \mathsf{U}_{\mathtt{Fn}}$ of functorial algebras.*

Proof. By Definition 5.1, we need to show that $K^{\mathtt{EM}}_{\mathtt{Fn}}$ commutes with left and right adjoints of both resolutions. For right adjoints, we have

$$\mathsf{U}_{\Sigma+\Gamma E}\langle X, \alpha\rangle = X = \Downarrow \mathsf{U}_{\mathtt{Fn}} K^{\mathtt{EM}}_{\mathtt{Fn}}\langle X, \alpha\rangle$$

and for left adjoints, we have

$$\begin{aligned}
K^{\mathtt{EM}}_{\mathtt{Fn}}(\mathtt{Free}_{\Sigma+\Gamma E} X) &= K^{\mathtt{EM}}_{\mathtt{Fn}}\langle (\Sigma + \Gamma E)^* X,\ \mathsf{op} : (\Sigma + \Gamma E)(\Sigma + \Gamma E)^* X \to (\Sigma + \Gamma E)X\rangle \\
&= \langle E, (\Sigma + \Gamma E)^* X, \mathsf{in}, \mathsf{op}\rangle \qquad \{E \cong G^*\underline{0} \text{ and } (\Sigma + \Gamma E) = I_E \text{ (Section 4.1)}\} \\
&\cong \langle G^*\underline{0}, (I_{G^*\underline{0}})^* X, \mathsf{op}^{G^*}_{\underline{0}}, \mathsf{op}^{(I_{G^*\underline{0}})^*}_X\rangle \qquad \{\text{By Lemma 4.2}\} \\
&\cong \mathtt{Free}_{\mathtt{Fn}}(\Uparrow X)
\end{aligned}$$

and similarly for the actions on morphisms. Here we only have $K^{\mathtt{EM}}_{\mathtt{Fn}}\mathtt{Free}_{\Sigma+\Gamma E}$ being isomorphic to $\mathtt{Free}_{\mathtt{Fn}} \Uparrow$ instead of a strict equality, since these two resolutions induce the monad E only up to isomorphism. To remedy this, one can generalise the definition of comparison functors to take an isomorphism into account, but we leave it out here since it is not very instructive. □

Remark 5.1. Now that we have comparison functors $K^{\mathtt{EM}}_{\mathtt{Fn}} : \mathbb{C}^E \to \mathtt{Fn\text{-}Alg}$ and $K^{\mathtt{Fn}}_{\mathtt{EM}} : \mathtt{Fn\text{-}Alg} \to \mathbb{C}^E$ that translate between EM algebras and functorial algebras in a way preserving the induced interpretation, thus their expressive power for interpretation is equivalent. This is not a contradiction to the fact that the categories $\mathbb{C}^E$ and $\mathtt{Fn\text{-}Alg}$ themselves are *not equivalent* (Remark 4.1): although functorial algebras have richer structures than EM ones, when we use them for interpretation, we only care about what we can observe in the base category. As an analogy, real numbers have richer structures than integers, but if we observe them by the floor function, they give the same set of observations.

5.4 Translating Functorial Algebras to Indexed Algebras

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

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

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 are $\langle a : \Sigma A \to A,\ d : \Gamma(\triangleleft A) \to A,\ p : A \to (\triangleleft A) \rangle$ given by

$$a_0 = (\alpha^I \cdot \iota_1) : \Sigma X \to X \qquad d_0 = (\alpha^I \cdot \iota_2) : \Gamma HX \to X$$
$$a_{i+1} = (\alpha^G_{H^i X} \cdot \iota_2) : \Sigma HH^i X \to H^{i+1} X \qquad d_{i+1} = (\alpha^G_{H^i X} \cdot \iota_3) : \Gamma HHH^i X \to H^{i+1} X$$

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

Lemma 5.4. *Functor $K^{\mathtt{Fn}}_{\mathtt{Ix}}$ is a comparison functor from the resolution $\mathtt{Free}_{\mathtt{Fn}} \Uparrow \dashv \Downarrow \mathtt{U}_{\mathtt{Fn}}$ of functorial algebras to the resolution $\mathtt{Free}_{\mathtt{Ix}} \restriction \dashv \lfloor \mathtt{U}_{\mathtt{Ix}}$ of indexed algebras.*

Proof. We need to show that $K^{\mathtt{Fn}}_{\mathtt{Ix}}$ satisfies the required commutativities

$$\Downarrow \mathtt{U}_{\mathtt{Fn}} \cong \lfloor \mathtt{U}_{\mathtt{Ix}} K^{\mathtt{Fn}}_{\mathtt{Ix}} \qquad \text{and} \qquad K^{\mathtt{Fn}}_{\mathtt{Ix}} \mathtt{Free}_{\mathtt{Fn}} \Uparrow \cong \mathtt{Free}_{\mathtt{Ix}} \restriction$$

in Definition 5.1 for it to be a comparison functor. First it is easy to see that it commutes with the right adjoints:

$$\lfloor \mathtt{U}_{\mathtt{Ix}} (K^{\mathtt{Fn}}_{\mathtt{Ix}} \langle H, X, \alpha^G, \alpha^I \rangle) = \lfloor \mathtt{U}_{\mathtt{Ix}} \langle A, a, d, p \rangle = A_0 = X = \lfloor \mathtt{U}_{\mathtt{Ix}} (K^{\mathtt{Fn}}_{\mathtt{Ix}} \langle H, X, \alpha^G, \alpha^I \rangle)$$

Its commutativity with the left adjoints is slightly more involved, and we show a sketch here. Piróg et al. [2018] show that $\mathtt{Free}_{\mathtt{Ix}} \restriction X$ is isomorphic to the indexed algebras carried by $E^+_X : \mathbb{C}^{|\mathbb{N}|}$ such that $(E^+_X)_i = E^{i+1} X$ with structure maps

$$k^{\Sigma}_n = \left(\Sigma (E^+_X)_n = \Sigma E E^n X \xrightarrow{\mathsf{in} \cdot \iota_2} E E^n X = (E^+_X)_n \right)$$
$$k^{\Gamma\triangleleft}_n = \left((\Gamma \triangleleft E^+_X)_n = \Gamma E E E^n X \xrightarrow{\mathsf{in} \cdot \iota_3} E E^n X = (E^+_X)_n \right)$$
$$k^{\triangleright}_n = \left((E^+_X)_n = E E^n X \xrightarrow{\mathsf{in} \cdot \iota_1} E E E^n X = (\triangleleft E^+_X)_n \right)$$

where $\mathsf{in} : Id + \Sigma E + \Gamma E E \to E$ is the isomorphism between E and GE. Also by Lemma 4.2, we know that $\mathtt{Free}_{\mathtt{Fn}} \Uparrow X$ is isomorphic to the functorial algebra $\langle E, EX, \mathsf{in}, [\mathsf{in} \cdot \iota_2, \mathsf{in} \cdot \iota_3] \rangle$. Clearly $K^{\mathtt{Fn}}_{\mathtt{Ix}} \mathtt{Free}_{\mathtt{Fn}} \Uparrow$ and $\mathtt{FreeIx} \restriction$ agree on the carrier E^+_X. It can be checked that they agree on the structure maps and the action on morphisms too. □

Since comparison functors preserve interpretation (Lemma 5.1), 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. Figure 3 summarises the comparison functors and resolutions that we have studied.

6 FUSION LAWS AND HYBRID FOLD

A crucial 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 and optimisation [Coutts et al. 2007; Hinze et al. 2011; Takano and Meijer 1995; Wadler 1988; Wu and Schrijvers 2015]. In this section, we show that the three adjunctions for the three kinds of algebras give us three fusion laws (Section 6.1), and the comparison functors constructed in the previous section allow us to switch from one kind of algebra to another whose fusion law is easier to work with. As a case study and another contribution of this paper, we use the fusion laws to show that the hybrid recursion scheme in [Piróg et al. 2018] is equivalent to interpreting with indexed algebras (Section 6.2).

6.1 Fusion Laws of Interpretation

Recall that given any resolution $L \dashv R$ of some monad $M : \mathbb{C} \to \mathbb{C}$ where $L : \mathbb{C} \to \mathbb{D}$, adjunction $L \dashv R$ corresponds to a natural isomorphism

$$\lfloor \cdot \rfloor : \mathbb{D}(LA, D) \cong \mathbb{C}(A, RD) : \lceil \cdot \rceil \tag{13}$$

and for any $g : A \to RD$, we have an interpretation morphism

$$eval_D\ g = R\lceil g \rceil = R(\epsilon_D \cdot Lg) : MA \to RD$$

Then if there is a morphism in the form of $(h \cdot eval_D\ g)$—an interpretation followed by some morphism—the following *fusion law* allows one to fuse it into a single interpretation.

Lemma 6.1 (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 any $D : \mathbb{D}$ and $h : RD \to X$, if there is some D' in $\mathbb{D}$ and $f : D \to D'$ such that $RD' = X$ and $Rf = h$, then*

$$h \cdot eval_D\ g = eval_{D'}\ (h \cdot g) \tag{14}$$

Proof. We have $h \cdot eval_D\ g = Rf \cdot R\lceil g \rceil = R(f \cdot \lceil g \rceil)$. Then by the naturality of (13) in D, $f \cdot \lceil g \rceil = \lceil Rf \cdot g \rceil$. Thus $R(f \cdot \lceil g \rceil) = R(\lceil Rf \cdot g \rceil) = R(\lceil h \cdot g \rceil) = eval_{D'}\ (h \cdot g)$. □

Applying the lemma to the three resolutions of E gives us three fusion laws: for any $D : \mathbb{D}$ where $\mathbb{D} \in \{\texttt{Ix-Alg}, \texttt{Fn-Alg}, \mathbb{C}^E\}$, one can can fuse $h \cdot eval_D\ g$ into a single interpretation if one can make h a $\mathbb{D}$-homomorphism.

Although the three kinds of algebras for interpreting scoped operations have the same expressivity theoretically, sometimes it is practically easier to make h an algebra homomorphism for a particular kind of algebras. The comparison functors in Section 5 allow one to switch from one kind of algebra to another that may be easier to work with: Let $K : \mathbb{D} \to \mathbb{D}'$ be a comparison functor, if there is some f in $\mathbb{D}'$ such that $R'f = h$, then

$$h \cdot eval_D\ g = h \cdot eval_{KD}\ g = eval_{FD}\ (h \cdot g)$$

In this case, we start with interpreting using $\mathbb{D}$-algebras and we switch to $\mathbb{D}'$ by comparison functor K and do the fusion in that category.

In the rest of this section, we demonstrate how the fusion laws and comparison functors can be used to prove that a recursion scheme in [Piróg et al. 2018], which we call a *hybrid fold*, coincides with interpretation with indexed algebras.

6.2 Case Study: Hybrid Fold

Although Piróg et al. [2018] propose the adjunction $\mathtt{Free}_{\mathtt{Ix}} \dashv \mathtt{U}_{\mathtt{Ix}}$ for interpreting scoped operations with indexed algebras, they use a recursive function *hfold* (Figure 1) different from *eval* in their Haskell implementation to interpret E with indexed algebras. Compared to a faithful implementation of *eval*, their *hfold* is more efficient since it skips transforming E to the free indexed algebra $\lfloor \mathtt{U}_{\mathtt{Ix}}\mathtt{Free}_{\mathtt{Ix}} \rfloor$ but directly interprets E with an indexed algebra. Thus we call it a *hybrid fold* since it works on a data structure E that does not directly match its type of indexed algebra.

While the definition of *hfold* is computationally intuitive, Piróg et al. [2018] do not provide a formal theory underlying the semantics of this recursive definition. In this subsection, we fill the gap by showing that *hfold* coincides with *eval* with indexed algebras. We divide the proof into three parts for clarity: after making the problem clear (Section 6.2.1), we first show that the *hfold* for an indexed algebra A is can be determined by a catamorphism from E in $\mathbb{C}^{\mathbb{C}}$ (Section 6.2.2), which is then shown to be a special case of interpreting with functorial algebras (Section 6.2.3), and finally we translate this functorial algebra into the category `Ix-Alg` of indexed algebras using $K^{\mathsf{Fn}}_{\mathtt{Ix}}$, and show that it induces the same interpretation as the one from the indexed algebra A that we start with (Section 6.2.4).

6.2.1 Semantic Problem of Hybrid Fold. Fix an indexed algebra carried by $A : \mathbb{C}^{|\mathbb{N}|}$ throughout this section

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

For notational convenience, we define a functor $S : |\mathbb{N}| \to |\mathbb{N}|$ such that $Sn = n + 1$, then $\triangleleft A = AS$ since $(\triangleleft A)_n = A_{n+1} = A(Sn)$. With functor S, we can view p and d as $p : A \to AS$ and $d : \Gamma AS \to A$. Then the recursive definition of *hfold* in Figure 1 can be understood as a morphism $h : EA \to A$ in $\mathbb{C}^{|\mathbb{N}|}$ satisfying the equation

$$h = [h_1, h_2, h_3] \cdot \mathsf{in}^\circ A \tag{15}$$

where $\mathsf{in}^\circ : E \to Id + \Sigma E + \Gamma EE$ is the isomorphism between E and GE, and h_1, h_2 and h_3 correspond to the three cases of *hfold* respectively:

$$h_1 = \left(IdA \xrightarrow{\mathsf{id}} A\right) \qquad h_2 = \left(\Sigma EA \xrightarrow{\Sigma h} \Sigma A \xrightarrow{a} A\right)$$

$$h_3 = \left(\Gamma EEA \xrightarrow{\Gamma Eh} \Gamma EA \xrightarrow{\Gamma Ep} \Gamma EAS \xrightarrow{\Gamma h} \Gamma AS \xrightarrow{d} A\right)$$

In general, an equation in the form of (15) does not necessarily have a solution or a unique solution. Thus the semantics of the recursively defined function *hfold* is obscure. We settle this problem with the following result.

Theorem 6.2 (Hybrid Folds Coincide with Interpretation)**.** *There exists a unique solution to (15) and it coincides with interpretation with indexed algebra A at level 0 (Equation 3):*

$$h_0 = eval_{\langle A,a,d,p\rangle}\ id : A_0 \to A_0$$

We prove the theorem in the rest of this section with the tools that we have developed.

6.2.2 Hybrid Fold Is an Adjoint Fold. The first step of our proof is to show the unique existence of the solution to (15) based on the observation that it is an *adjoint fold equation* [Hinze 2013] with the adjunction between *right Kan extension* [Hinze 2012; Mac Lane 1998] and composition with A. Hinze [2013] shows the following theorem stating that there is a unique solution to such adjoint fold equations.

Theorem 6.3 (Mendler-style Adjoint Folds [Hinze 2013])**.** *Given any functor $L : \mathbb{C} \to \mathbb{D}$ left adjoint to some $R : \mathbb{D} \to \mathbb{C}$, an endofunctor $G : \mathbb{D} \to \mathbb{D}$ whose initial algebra $\langle \mu G, in\rangle$ exists, and a natural*

transformation $\Phi : \mathbb{C}(L-, B) \to \mathbb{C}(LD-, B)$ *for some* $B : \mathbb{C}$*, then there exists a unique* $x : L(\mu G) \to B$ *satisfying*

$$x = \Phi_{\mu G}(x) \cdot Lin^\circ \tag{16}$$

and the unique solution satisfies $\lfloor x \rfloor = (\!| \lfloor \Phi_{RB}(\epsilon_B) \rfloor |\!)$ *where* $\lfloor \cdot \rfloor : \mathbb{C}(LD, C) \to \mathbb{D}(D, RC)$ *is the isomorphism for the adjunction* $L \dashv R$*.*

Since $h : EA \to A$ and $E = \mu G : \mathbb{C}^\mathbb{C}$, to apply this theorem to (15), we only need to (i) make $(-A) : \mathbb{C}^\mathbb{C} \to \mathbb{C}^{|\mathbb{N}|}$ a left adjoint and (ii) make $[h_1, h_2, h_3]$ in (15) an instance of $\Phi_E(h)$ for some natural transformation Φ:

- For (i), it is standard that the functor $-A$ is left adjoint to the *right Kan extension* along A [Hinze 2012; Mac Lane 1998], that is a functor $\mathsf{Ran}_A : \mathbb{C}^{|\mathbb{N}|} \to \mathbb{C}^\mathbb{C}$ which is usually constructed by the *end* formula $\mathsf{Ran}_A\ B\ X = \int_{n:|\mathbb{N}|} B_n^{\mathbb{C}(X, A_n)}$ or in the notation of type systems, polymorphic functions $\forall n.\ (X \to A\ n) \to B\ n$. Yet we do not need the explicit formulas for Ran_A in our proof, and we only need the properties of Ran_A as the right adjoint to $-A$.
- For (ii), we define a natural transformation $\Phi : \mathbb{C}^{|\mathbb{N}|}(-A, A) \to \mathbb{C}^{|\mathbb{N}|}((G-)A, A)$ such that for any $H : \mathbb{C}^\mathbb{C}$ and $f \in \mathbb{C}^{|\mathbb{N}|}(HA, A)$, $\Phi_H(f) = [f_1, f_2, f_3]$ where

$$f_1 = \left(IdA \xrightarrow{\mathsf{id}} A\right) \qquad f_2 = \left(\Sigma HA \xrightarrow{\Sigma f} \Sigma A \xrightarrow{a} A\right) \tag{17}$$
$$f_3 = \left(\Gamma HHA \xrightarrow{\Gamma Hf} \Gamma HA \xrightarrow{\Gamma Hp} \Gamma HAS \xrightarrow{\Gamma f} \Gamma AS \xrightarrow{d} A\right)$$

  and $\langle a, d, p\rangle$ are the structure maps carried by A. It is straightforward to check that (15) is exactly $h = \Phi_E(h) \cdot \mathsf{in}^\circ A$.

Then by Theorem 6.3 we have the following result.

Lemma 6.4 (Unique Existence of Hybrid Fold)**.** *The recursive definition* $h : EA \to A$ *(15) of hybrid folds has a unique solution if* $\mathsf{Ran}_A : \mathbb{C}^{|\mathbb{N}|} \to \mathbb{C}^\mathbb{C}$ *exists, and*

$$h = \left(EA \xrightarrow{(\!|\lfloor \Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor |\!)A} (\mathsf{Ran}_A A)A \xrightarrow{\epsilon_A} A\right) \tag{18}$$

where $\epsilon_A : (\mathsf{Ran}_A A)A \to A$ *is the counit of the adjunction* $(-A) \dashv \mathsf{Ran}_A$*, and* $(\!|\lfloor \Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor |\!)$ *is the catamorphism from the initial* G*-algebra* E *to the* G*-algebra carried by* $\mathsf{Ran}_A A$ *with structure map* $\lfloor \Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor$*.*

6.2.3 Catamorphism as Interpretation. We have shown that the hybrid fold of any indexed algebra A uniquely exists, but recall that we also want to show its coincidence with the interpretation morphism *eval* with the same indexed algebra A (the second part of Theorem 6.2). To this end, we show that the hybrid fold coincides with interpreting with some *functorial* algebra, and then use the comparison functor $K_{\mathsf{Ix}}^{\mathsf{Fn}}$ (Section 5.4) to translate this functorial algebra into an indexed algebra.

Recall that Lemma 6.4 states that the hybrid fold can be determined by a catamorphism induced by a G-algebra, which is very close to a functorial algebra. We have the following simple general result to relate them.

Lemma 6.5. *For any* G*-algebra* $\langle H : \mathbb{C}^\mathbb{C}, \alpha : GH \to H\rangle$ *and any* $X : \mathbb{C}$*, let* F *be the functorial algebra* $\langle H,\ HX,\ \alpha,\ \alpha_X \cdot [\iota_2, \iota_3] : \Sigma HX + \Gamma HHX \to HX\rangle$*. Then it holds that*

$$(\!|\alpha|\!)_X = eval_F\ (\alpha_X \cdot \iota_1) : EX \to HX$$

Proof. It directly follows from the fact that the formula (11) for computing *eval* with the functorial algebra F here is exactly the defining equation of the catamorphism $(\!|\alpha|\!)_X$. □

Now consider the G-algebra in Lemma 6.4 carried by $\mathsf{Ran}_A A$ with structure map $\alpha^G = \lfloor \Phi_{\mathsf{Ran}_A A}(\epsilon_A) \rfloor$. If we instantiate $X = A_0$ in Lemma 6.5, then we get

$$(\!|\alpha^G|\!)_{A_0} = eval_F\ (\alpha^G_{A_0} \cdot \iota_1)$$

where F is the following functorial algebra $\langle \mathsf{Ran}_A A, (\mathsf{Ran}_A A)A_0, \alpha^G, \alpha^G \cdot [\iota_2, \iota_3] \rangle$. Plugging the last equation into (18), we obtain

$$h_0 = (\epsilon_A)_0 \cdot eval_F\ (\alpha^G_{A_0} \cdot \iota_1) \tag{19}$$

Note that the right hand side takes exactly the form of an interpretation followed by a morphism $(\epsilon_A)_0 : (\mathsf{Ran}_A A)A_0 \to A_0$. Thus we try to use interpretation fusion (Lemma 6.1) to simplify it. Unsurprisingly, we consider the functorial algebra $F' = \langle \mathsf{Ran}_A A,\ A_0,\ \alpha^G,\ \alpha^I \rangle$ where

$$\alpha^I = [a_0 : \Sigma A_0 \to A_0, d_0 \cdot \Gamma(\epsilon_A)_0 \cdot \Gamma(\mathsf{Ran}_A A)p] : \Sigma A_0 + \Gamma(\mathsf{Ran}_A A)A_0 \to A_0$$

where $a : \Sigma A \to A$, $d : \Gamma AS \to A$ and $p : A \to AS$ are the structure maps of indexed algebra A. It can be checked that $\langle \mathsf{id}, (\epsilon_A)_0 \rangle$ is a functorial algebra homorphism from F to F'. Thus by Lemma 6.1, we obtain that

$$h_0 = eval_F\ \left((\epsilon_A)_0 \cdot (\alpha^G)_{A_0} \cdot \iota_1\right) = eval_{F'}\ \mathsf{id} : EA_0 \to A_0 \tag{20}$$

which means that the hybrid fold coincides with the interpretation with functorial algebra F'.

6.2.4 Translating Back to Indexed Algebras. The last step of our proof is translating the functorial algebra F' back to an indexed algebra using comparison functor $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ (Section 5.4), and showing that the resulting indexed algebra induces the same interpretation morphism as the one induced by A.

Recall that the comparison functor $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ maps a functorial algebra carried by $\langle H, X \rangle$ to an indexed algebra carried by $i \mapsto H^i X$. Thus $K^{\mathsf{Fn}}_{\mathsf{Ix}} F'$ is an indexed algebra carried by $i \mapsto (\mathsf{Ran}_A A)^i A_0$ and by Lemma 5.1 and (20), $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ preserves the induced interpretation:

$$h_0 = eval_{F'}\ \mathsf{id} = eval_{K^{\mathsf{Fn}}_{\mathsf{Ix}} F'}\ \mathsf{id} : EA_0 \to A_0 \tag{21}$$

What remains is to prove that $eval_{K^{\mathsf{Fn}}_{\mathsf{Ix}} F'}\ \mathsf{id} = eval_A\ \mathsf{id}$, and we show this by interpretation fusion again: we define a morphism τ from $i \mapsto (\mathsf{Ran}_A A)^i A_0$, which is the carrier of $K^{\mathsf{Fn}}_{\mathsf{Ix}} F'$, to A in $\mathbb{C}^{|\mathbb{N}|}$ such that $\tau_0 = \mathsf{id} : (\mathsf{Ran}_A A)^0 A_0 \to A_0$ and

$$\tau_{i+1} = \left((\mathsf{Ran}_A A)(\mathsf{Ran}_A A)^i A_0 \xrightarrow{(\mathsf{Ran}_A A)\tau_i} (\mathsf{Ran}_A A)A_i \xrightarrow{(\mathsf{Ran}_A A)p} (\mathsf{Ran}_A A)A_{i+1} \xrightarrow{\epsilon_A} A_{i+1}\right)$$

and it can be checked that τ is an indexed algebra homomorphism from $K^{\mathsf{Fn}}_{\mathsf{Ix}} F'$ to $\langle A, a, d, p \rangle$. Note that we have $\lfloor \mathsf{U}_{\mathsf{Ix}} \tau$ equals $\mathsf{id} : A_0 \to A_0$, and by Lemma 5.1, we have that

$$h_0 = eval_{K^{\mathsf{Fn}}_{\mathsf{Ix}} F'}\ \mathsf{id} = \mathsf{id} \cdot eval_{K^{\mathsf{Fn}}_{\mathsf{Ix}} F'}\ \mathsf{id} = (\lfloor\ \mathsf{U}_{\mathsf{Ix}})\tau \cdot eval_{K^{\mathsf{Fn}}_{\mathsf{Ix}} F'}\ \mathsf{id} = eval_A\ \mathsf{id}$$

This completes our proof of Theorem 6.2.

Remark 6.1. We only stated in Theorem 6.2 that the hybrid fold coincides with *eval* at level 0 since it simplifies the proof and in practice usually only the component h_0 is used by the programmer once h is defined. Yet it is straightforward to generalise the proof above to prove that for any $n : |\mathbb{N}|$,

$$h_n = eval_{\triangleleft^n A}\ \mathsf{id} : EA_n \to A_n$$

where $\triangleleft^n A$ is the indexed algebra obtained from A by shifting the indices by n: $(\triangleleft^n A)_i = A_{n+i}$.

Remark 6.2. This proof demonstrates the flexibility that we get by having more than one type of algebras to interpret scoped operations, provided with comparison functors to switch between them: in the proof, we first use interpretation fusion in the category of functorial algebras, and translate it to the category of indexed algebras by $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ and use another interpretation fusion there.

7 RELATED WORK

The most closely related work is of course that of Piróg et al. [2018] on categorical models of scoped effects. That work in turn builds on Wu et al. [2014] who introduced the notion of scoped effects after identifying modularity problems with using algebraic effect handlers for catching exceptions [Plotkin and Pretnar 2013]. Scoped effects have found their way into several Haskell implementations of algebraic effects and handlers [King 2019; Maguire 2019; Rix et al. 2018].

Effect Handlers and Modularity. Moggi [1989] and Wadler [1990] popularized monads for respectively modeling and programming with computational effects. Soon after, the desire arose to define complex monads by combining modular definitions of individual effects [Jones and Duponcheel 1993; Steele 1994], and monad transformers were developed to meet this need [Liang et al. 1995].

Yet, several years later, algebraic effects were proposed as an alternative more structured approach for defining and combining computational effects [Hyland et al. 2006; Plotkin and Power 2002, 2003]. The addition of handlers [Plotkin and Pretnar 2013] has made them practical for implementation and many languages and libraries have been developed since. Schrijvers et al. [2019] have characterized modular handlers by means of modular carriers, and shown that they correspond to a subclass of monad transformers. Forster et al. [2019] have also shown that algebraic effects, monads and delimited control are macro-expressible in terms of each other. It would be interesting to extend these investigations and formally position the expressivity of scoped effects with respect to these other approaches.

Scoped operations are generally not algebraic operations in the original design of algebraic effects [Plotkin and Power 2002], but as we have seen in Section 3, an alternative view on Eilenberg-Moore algebras of scoped operations is regarding them as handlers of *algebraic* operations of signature $(\Sigma + \Gamma E)$. Note that the functor $(\Sigma + \Gamma E)$ mentions the type E modelling computations, and thus it is not a valid signature of algebraic effects in the original design of effect handlers [Plotkin and Pretnar 2009, 2013], 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 [Bauer and Pretnar 2014], Koka [Leijen 2017] and Frank [Lindley et al. 2017] 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. In particular, languages supporting *shallow handlers* [Hillerström and Lindley 2018] like Frank are a good fit for this purpose, since they allow general recursion and case splits over the type E of computations, which is usually needed to concisely implement the component $\Gamma EX \to X$ of an Eilenberg-Moore algebra of scoped operations.

Other variations of scoped effects have recently been suggested. Recently, Poulsen et al. [2021] 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. In Ahman and Pretnar [2021] the authors investigate *asynchronous effects*. The authors note that asynchronous effects are in fact *scoped* algebraic 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 programming language. The benefit of abstract syntax is that it allows for *generic programming* in programming languages like Haskell that have support for, e.g. type classes, GADTs [Johann and Ghani 2008] and so on. As an example, Swierstra [2008] showed that it is possible to modularly create compilers by formalising syntax with Haskell data types.

The problem of formalising abstract syntax for languages with variable binding was first addressed by Fiore et al. [1999]; Fiore and Turi [2001]. Subsequently, Ghani et al. [2006] model the abstract syntax of explicit substitutions as an initial algebra in the endofunctor category and show that it is a monad. In this paper we use a monad E, which is a slight generalisation of the monad of explicit substitutions, to model the syntax of scoped operations. Ghani et al. [2006] also comment on the apparent lack of modularity in their approach in the sense that the monad representing a combined language cannot be decomposed by the coproduct of the monads representing its sub-languages. Piróg et al. [2018] point out the resolution $\mathtt{Free}_{\mathtt{Ix}} \restriction \dashv \rfloor\, \mathtt{U}_{\mathtt{Ix}}$ of E via indexed algebras restores some modularity in the sense that

$$E \;\cong\; \lfloor\, (\Sigma - + \Gamma(\triangleleft -) + (\triangleright -))^* \,\restriction \;\cong\; \lfloor\, (\Sigma^* +_{\mathtt{Mnd}} (\Gamma(\triangleleft -) + (\triangleright -))^*) \,\restriction$$

where $+_{\mathtt{Mnd}}$ is the coproduct in the category of monads on $\mathbb{C}^{|\mathbb{N}|}$. This kind of modularity also holds for the resolution via functorial algebras, but we have not yet explored using this form of modularity to implement modular interpretation of scoped effects.

Hyland et al. [2006] develop uniform ways to combine algebraic effects presented as Lawvere theories or equational theories, such as by adding commutativity equations of operations. In this paper, we have not considered equations of scoped operations. A possible direction towards this is the framework of second-order algebraic operations by Fiore and Mahmoud [2010].

Recursion schemes and Adjoint Folds. Recursion schemes have received a considerable amount of attention. Of particular relevance to us are recursion schemes for nested datatypes, which were first investigated by Bird and Paterson [1999], who added some extra parameters to the natural transformations. This was later given a categorical treatment by Johann and Ghani [2007], who describe initial algebras of endofunctors as a way to formalise recursion schemes for nested data types under the slogan "Initial algebras are enough". We support this view by using the higher-order endofunctor G to define the monad E that models syntax of scoped operations.

The study of nested types gave rise to the adjoint folds of Hinze [2013], who provided the framework that our construction follows closely. This work introduces adjoints folds of form $L\mu D \longrightarrow A$ for some adjunction $L \dashv R$ as a generalisation of catamorphisms $\mu D \longrightarrow A$ that encompass many recursion schemes such as mutumorphisms, paramorphisms, and recursion schemes from comonads [Hinze and Wu 2016], thus supporting the view that "adjoint functors arise everywhere". In comparison, our interpretation scheme $eval_D\ g : GFX \longrightarrow A$ for some adjunction $F \dashv G$ can be understood as another way of generalising catamorphisms $(\mu D \cong) \mathtt{U}_D\mathtt{Free}_D 0 \longrightarrow A$ where the free-forgetful adjunction is replaced by an arbitrary resolution $F \dashv G$. These two generalisations are orthogonal, and one can consider *adjoint interpretations* $LGFX \longrightarrow A$ to implement more sophisticated forms of handlers of scoped operations, such as parameterised handlers [Leijen 2017] and multihandlers [Lindley et al. 2017].

We also show that the hybrid fold recursion scheme in Piróg et al. [2018] is an adjoint fold with the adjunction of right Kan extension, and that it coincides with interpretation with indexed algebras. As future work we wish to consider if there is a systematic way of obtaining such hybrid recursion schemes as efficient implementations of interpretations with other kinds of algebras.

Step-Indexing. It is interesting to note here that the operators $\triangleright$ and $\triangleleft$ used here and first discovered in Piróg et al. [2018] are very similar to $\blacktriangleright$ and $\blacktriangleleft$ (also pronounced "later" and "previous") used in *guarded recursion* [Birkedal et al. 2012]. The intended meaning is that $\triangleright$ and $\blacktriangleright$ decrease the index while $\triangleleft$ and $\blacktriangleleft$ increase it, but while $\triangleright$ and $\triangleleft$ live in $\mathbb{C}^{|\mathbb{N}|}$, the guarded recursive counterparts live in the category of presheaves over ω, namely $\mathtt{Set}^{\omega^{\mathrm{op}}}$. It is also interesting to note that the type $MA \cong A + \triangleright MA$ is very similar to the *partiality monad* from Atkey and McBride [2013]. However, it is not very clear to what extent these two areas of research connect.

8 CONCLUSION

Motivated by the implementation difficulty of indexed algebras, this paper presented two alternative ways of interpreting scoped operations, namely, Eilenberg-Moore algebras and functorial algebras, which trade off structuredness for simplicity in different ways, and thus can be implemented with more modest programming language features, as demonstrated in our Haskell and OCaml implementations and programming examples.

We put the three models of interpreting scoped operations in the same framework of resolutions of the monad modelling syntax, and by constructing interpretation-preserving functors between the three kinds of algebras, we showed that they have equivalent expressivity for interpreting scoped operation, although they form non-equivalent categories. The uniform theoretical framework also gave rise to fusion laws of interpretation in a straightforward way, which we used to provide a solid theoretical foundation for the hybrid fold recursion scheme in [Piróg et al. 2018] which was previously missing.

There are two strains of work that should be pursued from here. The first one would be investigating modularity of algebras of scoped operations, in particular, ways to *forward* scoped operations that is not handled. The second one would be the design of a language supporting handlers of scoped operations natively and its type system and operational semantics.

REFERENCES

Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. *Proc. ACM Program. Lang.* POPL (2021). https://doi.org/10.1145/3434305

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.).

Michael Barr. 1970. Coequalizers and free triples. *Mathematische Zeitschrift* (1970).

Andrej Bauer and Matija Pretnar. 2014. 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

Richard S. Bird and Ross Paterson. 1999. Generalised folds for nested datatypes. *Formal Aspects Comput.* (1999). https://doi.org/10.1007/s001650050047

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* (2012).

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. *SIGPLAN Not.* 42, 9 (Oct. 2007), 315–326. https://doi.org/10.1145/1291220.1291199

Marcelo Fiore and Ola Mahmoud. 2010. Second-Order Algebraic Theories. In *Mathematical Foundations of Computer Science 2010*, Petr Hliněný and Antonín Kučera (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–380.

Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*.

Marcelo P. Fiore and Daniele Turi. 2001. Semantics of Name and Value Passing. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15. https://doi.org/10.1017/S0956796819000121

Neil Ghani, Tarmo Uustalu, and Makoto Hamana. 2006. Explicit substitutions and higher-order syntax. *High. Order Symb. Comput.* (2006).

J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1977. Initial Algebra Semantics and Continuous Algebras. *J. ACM* 24, 1 (Jan. 1977), 68–95. https://doi.org/10.1145/321992.321997

T. Hagino. 1987. *Category Theoretic Approach to Data Types.* Ph.D. Dissertation. University of Edinburgh.

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. *Lecture Notes in Computer Science* 11275 LNCS (2018), 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–362. https://doi.org/978-3-642-31113-0_16

Ralf Hinze. 2013. Adjoint folds and unfolds—An extended study. *Science of Computer Programming* (2013).

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

Ralf Hinze and Nicolas Wu. 2016. Unifying structured recursion schemes - An Extended Study. *J. Funct. Program.* (2016).

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining Effects: Sum and Tensor. *Theor. Comput. Sci.* 357, 1 (July 2006), 70–99. https://doi.org/10.1016/j.tcs.2006.03.013

Patricia Johann and Neil Ghani. 2007. Initial Algebra Semantics Is Enough!. In *Typed Lambda Calculi and Applications, TLCA (Lecture Notes in Computer Science)*. Springer. https://doi.org/10.1007/978-3-540-73228-0_16

Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 297–308. https://doi.org/10.1145/1328438.1328475

Mark P. Jones and Luc Duponcheel. 1993. *Composing Monads.* Research Report YALEU/DCS/RR-1004. Yale University, New Haven, Connecticut, USA. http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf

Alexis King. 2019. eff – screaming fast extensible effects for less. https://github.com/hasura/eff.

J. Lambek and P. J. Scott. 1986. *Introduction to Higher Order Categorical Logic.*

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

Saunders Mac Lane. 1998. *Categories for the Working Mathematician, 2nd edn.* Springer, Berlin.

Sandy Maguire. 2019. polysemy: Higher-order, low-boilerplate free monads. https://hackage.haskell.org/package/polysemy.

Eugenio Moggi. 1989. *An Abstract View of Programming Languages.* Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.).

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference (FOSSACS 2002)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec 2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. 2021. Staged Effects and Handlers for Modular Languages with Abstraction. To Appear.

Emily Riehl. 2017. *Category Theory in Context.* Dover Publications.

Rob Rix, Patrick Thomson, Nicolas Wu, and Tom Schrijvers. 2018. fused-effects: A fast, flexible, fused effect system. https://hackage.haskell.org/package/fused-effects.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. 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.* 98–113. https://doi.org/10.1145/3331545.3342595

Guy L. Steele, Jr. 1994. Building Interpreters by Composing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '94)*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM, 472–492. https://doi.org/10.1145/174675.178068

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Akihiko Takano and Erik Meijer. 1995. 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. https://doi.org/10.1145/224164.224221
Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A
Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. ACM, 61–78. https://doi.org/10.1145/91556.91592
Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 302–322. https://doi.org/978-3-319-19797-5_15
Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *SIGPLAN Not.* (2014).