Structured Handling of Scoped Effects

ANONYMOUS AUTHOR(S)

Algebraic effects offer a versatile framework that covers a wide variety of effects. However, an important family of operations—those that delimit a scope, and usually modelled as handlers—are not algebraic, thus preventing them from being used freely in conjunction with algebraic operations. Although proposals for scoped operations exist, they are either ad-hoc and unprincipled, or too inconvenient for practical programming. This paper provides the best of both worlds: a theoretically-founded model of scoped effects that is convenient for implementation and reasoning.

Our new model is based on an adjunction between a locally finitely presentable category and a category of *functorial algebras*. Using comparison functors between adjunctions, we show that our new model, the earlier indexed model, and another approach by simulating scoped operations with algebraic operations have equal expressivity for handling scoped operations. We consider our new model to be the sweet spot between ease of implementation and structuredness. Additionally, our approach automatically induces fusion laws of handlers of scoped effects, which are useful for reasoning and optimisation. As a case study, we use them to show the correctness of a hybrid recursion scheme, which was previously unclear. Our theoretical development is accompanied by practical examples of scoped operations in both Haskell and OCaml.

1 INTRODUCTION

For a long time monads [Moggi 1991; Spivey 1990; Wadler 1995] 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* [Plotkin and Power 2002], is gaining more traction. A big breakthrough has been the introduction of *handlers* [Plotkin and Pretnar 2013], which has made algebraic effects suitable for programming and has led to numerous dedicated languages and libraries implementing algebraic effects and handlers. In comparison to monads, algebraic effects provide a more modular approach to computations with effects, in which the syntax and semantics of effects are separated—computations invoking algebraic operations can be defined syntactically, and the semantics of operations are given by handlers separately in possibly many ways.

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

To remedy the situation, Wu et al. [2014] proposed a practical, but ad-hoc, generalization of algebraic effects in Haskell that encompasses scoped effects, that has been adopted by several algebraic effects libraries [King 2019; Maguire 2019; Rix et al. 2018]. More recently, Piróg et al. [2018] sought to put this ad-hoc approach for scoped effects on the same formal footing as algebraic effects. Their solution resulted in a construction based on a level-indexed category, called *indexed algebras*, as the way to give semantics to scoped effects. However, this formalization introduces a disparity between syntax and semantics that makes indexed algebras not as structured as the programs they

2020. 2475-1421/2020/1-ART427 $15.00
https://doi.org/

interpret, where they use an ad-hoc hybrid fold that requires indexing for the handlers, but not for the program syntax. Moreover, indexed algebras are not ideal for widespread implementation as they require dependent typing, in at least a limited form like GADTs [Johann and Ghani 2008].

This paper presents the principles of programming languages with scoped effects. To do so we introduce handlers based on *functorial algebras* that are both principled and formally grounded, and at the same time more structured than using the indexed algebras of Piróg et al. [2018], Additionally, our approach can be practically implemented without the need for dependent types or GADTs, making it available for a wider range of programming languages. In particular, after introducing the appropriate background on scoped effects, we make the following contributions:

- We motivate functorial algebras and demonstrate an implementation in Haskell, along with a number of programming examples to provide some intuition (Section 2);
- We develop a categorical foundation of functorial algebras as a notion of handlers of scoped effects. Specifically, we show that there is an adjunction between functorial algebras and a base category, inducing the monad modelling the syntax of scoped effects (Section 3);
- We show that the expressivity of functorial algebras, Piróg et al. [2018]'s indexed algebras, and simulating scoped effects with algebraic operations and recursion are equal, by constructing interpretation-preserving functors between the three approaches (Section 4);
- We present the fusion law of functorial algebras and use this to prove that the hybrid fold recursion scheme of Piróg et al. [2018] is correct, putting it on solid formal footing (Section 5).

Finally, we discuss related work (Section 6) and conclude (Section 7).

2 SCOPED EFFECTS FOR THE WORKING PROGRAMMER

In this section, we first show a motivating example of scoped effects and the loss of modularity when modelling them as effect handlers (Section 2.1), and then we demonstrate how the problem is solved by modelling scoped operations syntactically and handling them with *functorial algebras* in Haskell (Section 2.2), whose formal foundation will be developed in the rest of this paper.

2.1 Non-Modularity of Scoped Operations as Handlers

For the purpose of demonstration, assume that we are working with a Haskell library implementing algebraic effects and handlers, e.g. using *free monads* or *freer monads* [Kiselyov and Ishii 2015]. We are intentionally non-specific about how the implementation works, since the ideas that we are trying to convey are not confined to some particular implementations of effect handlers.

In this setting, signatures of algebraic operations are represented as Haskell functors Σ, and there is a monad M_Σ representing computations invoking operations in Σ. For example, let *ES* be the signature of algebraic operations for *exceptions* and a *mutable Int-state*, that provides the operations

$$\mathit{throw} :: M_{ES}\ a \qquad \mathit{put} :: \mathit{Int} \to M_{ES}\ () \qquad \mathit{get} :: M_{ES}\ \mathit{Int}$$

for throwing an exception, writing the mutable state, and reading the state respectively. With these operations, we can write programs such as updating the state s to n / s for some $n :: \mathit{Int}$ and throwing an exception when s is 0:

$$\begin{aligned}&\mathit{divideByState} :: \mathit{Int} \to M_{ES}\ \mathit{Int}\\ &\mathit{divideByState}\ n = \mathbf{do}\ \{s \leftarrow \mathit{get}; \mathbf{if}\ s \equiv 0\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{put}\ (n\ /\ s); \mathit{return}\ (n\ /\ s)\}\end{aligned}$$

where the **do**-notation means sequential composition of computations. Such a program is entirely syntactic, and to give it meaning a *handler* is applied, which takes a program of type $M_\Sigma\ a$ and interpets its values and operations into a type b.

$$\mathit{handle} :: (\Sigma\ b \to b) \to (a \to b) \to M_\Sigma\ a \to b$$

The first argument, of type $\Sigma\ b \longrightarrow b$, is used to interpret the operations and is called a Σ-*algebra* for the type b, where b is called the *carrier type* of the handler. The second argument, of type $a \longrightarrow b$, transforms the value returned by the computation into the carrier type b.

For example, given a term $r :: ES\ (M_{ES}\ a)$ for some a, an algebra $catchHdl\ r :: ES\ (M_{ES}) \longrightarrow M_{ES}$ for a handler that deals with *throw* is

$$\begin{aligned} &catchHdl :: M_{ES}\ a \longrightarrow ES\ (M_{ES}\ a) \longrightarrow M_{ES}\ a \\ &catchHdl\ r\ Throw = r \\ &catchHdl\ r\ op \quad\ \ = call\ op \end{aligned} \tag{1}$$

which evaluates r for *r*ecovery in case of an exception (where *Throw* corresponds to the constructor for the *throw* operation), and signals that other operations are left unhandled by using $call :: \Sigma\ (M_\Sigma\ a) \longrightarrow M_\Sigma\ a$ (provided by the library). An important advantage of the approach of effect handlers is that different semantics of a computation can be given by different handlers. For example, suppose that in some scenario one would like to interpret exceptions as unrecoverable errors and stop the execution of the program when an exception is raised. Then the following handler can be defined for this behaviour:

$$\begin{aligned} &catchHdl' :: M_{ES}\ a \longrightarrow ES\ (M_{ES}\ (Maybe\ a)) \longrightarrow M_{ES}\ (Maybe\ a) \\ &catchHdl'\ r\ Throw = return\ Nothing \\ &catchHdl'\ r\ op \quad\ \ = call\ op \end{aligned} \tag{2}$$

As expected, applying these two handlers to the program *divideByState* 5 produces different results:

$$\begin{aligned} &\quad handle\ (catchHdl\ (return\ 42))\ return\ (divideByState\ 5) \qquad\qquad :: M_{ES}\ Int \\ &= \mathbf{do}\ \{s \longleftarrow get; \mathbf{if}\ s \equiv 0\ \mathbf{then}\ return\ 42\ \mathbf{else}\ put\ (n\ /\ s); return\ (n\ /\ s)\} \\ &\quad handle\ (catchHdl'\ (return\ 42))\ (return \cdot Just)\ (divideByState\ 5) \quad :: M_{ES}\ (Maybe\ Int) \\ &= \mathbf{do}\ \{s \longleftarrow get; \mathbf{if}\ s \equiv 0\ \mathbf{then}\ return\ Nothing\ \mathbf{else}\ put\ (n\ /\ s); return\ (Just\ (n\ /\ s))\} \end{aligned}$$

Handlers generally give a semantics to syntactic algebraic operations, and in cases such as *catch* in the example above, they also model non-algebraic operations. This leads to a somewhat subtle complication: as observed by Wu et al. [2014], when non-algebraic operations (such as *catch*) are modelled with handlers, these handlers play a dual role of (i) modelling the syntax of the operation (the syntax for *catch* is the scope for which exceptions are caught) and (ii) giving semantics to it.

To see the problem more concretely, ideally one would like to have an (syntactic) operation

$$catch :: M_{ES}\ a \longrightarrow M_{ES}\ a \longrightarrow M_{ES}\ a$$

that acts on computations without giving semantics a priori, allowing to write programs like

$$prog = \mathbf{do}\ \{x \longleftarrow catch\ (divideByState\ 5)\ (return\ 42); put\ (x+1)\} \tag{3}$$

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

$$\begin{aligned} &\qquad \mathbf{do}\ \{x \longleftarrow handle\ (catchHdl\ (return\ 42))\ return\ (divideByState\ 5); put\ (x+1)\} \\ &\text{vs.}\quad \mathbf{do}\ xMb \longleftarrow handle\ (catchHdl'\ (return\ 42))\ (return \cdot Just)\ (divideByState\ 5) \\ &\qquad\quad \mathbf{case}\ xMb\ \mathbf{of}\ \{Nothing \longrightarrow return\ Nothing; (Just\ x) \longrightarrow put\ (x+1)\} \end{aligned}$$

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

2.2 Scoped Effects and Functorial Algebras

With the problem recognised, now we show a sketch of an alternative approach that we will develop in the rest of this paper to modelling a family of non-algebraic operations called *scoped operations* [Piróg et al. 2018], and we show how our approach solves the problem of modelling exception catching. For concreteness, we present our solution in Haskell in this section, and the formal categorical foundation of our approach will be studied in the next sections.

Syntax of Scoped Operations. To achieve modular interpretation of (non-algebraic) operations delimiting scopes, such as *catch* for exception handling and *spawn* p q for parallel composition of current programs, we model programs invoking algebraic operations and scoped operations *syntactically* by a monad *Prog* Σ Γ. The monad is parameterised by two Haskell functors Σ and Γ, which we call *signature functors* for algebraic operations and scoped operations respectively. The intuition is that a function $o :: (R \to x) \to \Sigma\ x$ represents an algebraic operation that produces a result of type R, or equivalently, has R-many possible ways to continue the computation after the operation. By contrast, a function $(N \to x) \to \Gamma\ x$ represents a scoped operation that creates N-many scopes enclosing programs.

Example 2.1. As we have seen in the previous subsection, an effect of *exceptions* has an algebraic operation for *throwing* exceptions, which produces no values, and also a scoped operation for *catching* exceptions, which creates two scopes, one enclosing the program for which exceptions are caught, and the other enclosing the recovery computation. Thus the algebraic and scoped signature functors for exceptions are respectively

$$\textbf{data}\ \mathit{Throw}\ x = \mathit{Throw} \qquad\qquad \textbf{data}\ \mathit{Catch}\ x = \mathit{Catch}\ x\ x \tag{4}$$

Example 2.2. An effect of *explicit nondeterminism* has two algebraic operations for nondeterministic *choice* and a scoped operation *Once*:

$$\textbf{data}\ \mathit{Choice}\ x = \mathit{Fail} \mid \mathit{Or}\ x\ x \qquad\qquad \textbf{data}\ \mathit{Once}\ x = \mathit{Once}\ x \tag{5}$$

The intuition is that the effect implements logic programming [Hinze 1998]—solutions to a problem are exhaustively searched: operation *Or* p q splits a searching branch into two; *Fail* marks a branch failed; and the scoped operation *Once* p keeps only the first solution found by p, making it *semi-deterministic*, which is useful for speed up searching with heuristics from the programmer.

Given signature functors Σ and Γ, the syntax trees *Prog* Σ Γ a of programs with algebraic operations Σ and scoped operations Γ are modelled by the following datatype:

$$\textbf{data}\ \mathit{Prog}\ \Sigma\ \Gamma\ a = \mathit{Return}\ a \mid \mathit{Call}\ (\Sigma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a)) \mid \mathit{Enter}\ (\Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ (\mathit{Prog}\ \Sigma\ \Gamma\ a))) \tag{6}$$

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

$$\begin{array}{l}\textbf{instance}\ (\mathit{Functor}\ \Sigma, \mathit{Functor}\ \Gamma) \Rightarrow \mathit{Monad}\ (\mathit{Prog}\ \Sigma\ \Gamma)\ \textbf{where} \\ \quad \mathit{return}\ a = \mathit{Return}\ a \qquad\qquad (\mathit{Call}\ \mathit{op}) \gg\!= k = \mathit{Call}\ (\mathit{fmap}\ ({\gg\!=}k)\ \mathit{op}) \\ \quad (\mathit{Return}\ a) \gg\!= k = k\ a \qquad (\mathit{Enter}\ \mathit{sc}) \gg\!= k = \mathit{Enter}\ (\mathit{fmap}\ (\mathit{fmap}\ ({\gg\!=}k))\ \mathit{sc})\end{array} \tag{7}$$

$$\begin{array}{ll}
\textbf{data}\ \mathit{EndoAlg}\ \Sigma\ \Gamma\ f = & \textbf{data}\ \mathit{BaseAlg}\ \Sigma\ \Gamma\ f\ a = \\
\quad \mathit{EndoAlg}\ \{\mathit{returnE} :: \forall x.\ x \to f\ x & \quad \mathit{BaseAlg}\ \{\mathit{callB} :: \Sigma\ a \to a \\
\qquad ,\ \mathit{callE} :: \forall x.\ \Sigma\ (f\ x) \to f\ x & \qquad ,\ \mathit{enterB} :: \Gamma\ (f\ a) \to a\} \\
\qquad ,\ \mathit{enterE} :: \forall x.\ \Gamma\ (f\ (f\ x)) \to f\ x\} &
\end{array}$$

$$\begin{array}{l}
\mathit{hcata} :: (\mathit{Functor}\ \Sigma, \mathit{Functor}\ \Gamma) \Rightarrow (\mathit{EndoAlg}\ \Sigma\ \Gamma\ f) \to \mathit{Prog}\ \Sigma\ \Gamma\ a \to f\ a \\
\mathit{hcata}\ \mathit{alg}\ (\mathit{Return}\ x) \quad = \mathit{returnE}\ \mathit{alg}\ x \\
\mathit{hcata}\ \mathit{alg}\ (\mathit{Call}\ \mathit{op}) \quad\ = (\mathit{callE}\ \mathit{alg} \cdot \mathit{fmap}\ (\mathit{hcata}\ \mathit{alg}))\ \mathit{op} \\
\mathit{hcata}\ \mathit{alg}\ (\mathit{Enter}\ \mathit{scope}) = (\mathit{enterE}\ \mathit{alg} \cdot \mathit{fmap}\ (\mathit{hcata}\ \mathit{alg} \cdot \mathit{fmap}\ (\mathit{hcata}\ \mathit{alg})))\ \mathit{scope}
\end{array}$$

$$\begin{array}{l}
\mathit{handle} :: (\mathit{Functor}\ \Sigma, \mathit{Functor}\ \Gamma) \Rightarrow (\mathit{EndoAlg}\ \Sigma\ \Gamma\ x) \to (\mathit{BaseAlg}\ \Sigma\ \Gamma\ x\ b) \to (a \to b) \to \mathit{Prog}\ \Sigma\ \Gamma\ a \to b \\
\mathit{handle}\ \mathit{ealg}\ \mathit{balg}\ \mathit{gen}\ (\mathit{Return}\ x) \quad = \mathit{gen}\ x \\
\mathit{handle}\ \mathit{ealg}\ \mathit{balg}\ \mathit{gen}\ (\mathit{Call}\ \mathit{op}) \quad\ = (\mathit{callB}\ \mathit{balg} \cdot \mathit{fmap}\ (\mathit{handle}\ \mathit{ealg}\ \mathit{balg}\ \mathit{gen}))\ \mathit{op} \\
\mathit{handle}\ \mathit{ealg}\ \mathit{balg}\ \mathit{gen}\ (\mathit{Enter}\ \mathit{scope}) = (\mathit{enterB}\ \mathit{balg} \cdot \mathit{fmap}\ (\mathit{hcata}\ \mathit{ealg} \cdot \mathit{fmap}\ (\mathit{handle}\ \mathit{ealg}\ \mathit{balg}\ \mathit{gen})))\ \mathit{scope}
\end{array}$$

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

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

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

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

For example, the smart constructors for the effect of exceptions (Example 2.1) are

$$\begin{array}{ll}
\mathit{throw} :: \mathit{Prog}\ \mathit{Throw}\ \Gamma\ a & \mathit{catch} :: \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \to \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \to \mathit{Prog}\ \Sigma\ \mathit{Catch}\ a \\
\mathit{throw} = \mathit{Call}\ \mathit{Throw} & \mathit{catch}\ h\ r = \mathit{Enter}\ (\mathit{Catch}\ (\mathit{fmap}\ \mathit{return}\ h)\ (\mathit{fmap}\ \mathit{return}\ r))
\end{array}$$

With smart constructors of the relevant signatures, now we can define the program (3) that we wanted to write, in which scoped operations and algebraic operations are modelled syntactically:

$$\begin{array}{l}
\mathit{prog} :: \mathit{Prog}\ (\mathit{Throw} + \mathit{State})\ \mathit{Catch}\ () \\
\mathit{prog} = \textbf{do}\ \{x \leftarrow \mathit{catch}\ (\mathit{divideByState}\ 5)\ (\mathit{return}\ 42); \mathit{put}\ (x+1)\}
\end{array}$$

Handlers of Scoped Operations. The *Prog* monad merely models the syntax of effectful computations, and their semantics need to be given by handlers. The goal of this paper is precisely developing a kind of handlers for scoped operations, which we call *functorial algebras*.

Given signatures Σ and Γ for algebraic and scoped operations, a functorial algebra for them is a quadruple $\langle f, b, \mathit{ealg}, \mathit{balg}\rangle$ for some functor f called the *functor carrier*, type b called the *base carrier*, and the other two components have the following types shown in Figure 1:

$$\mathit{ealg} :: \mathit{EndoAlg}\ \Sigma\ \Gamma\ f \qquad\qquad \mathit{balg} :: \mathit{BaseAlg}\ \Sigma\ \Gamma\ f\ b$$

The intuition is that functor f and *ealg* interpret the part of a program enclosed by scoped operations, and the type b and *balg* interpret the part of a program not enclosed by any scopes.

Example 2.3. The standard semantics of exception catching (cf. handler (1)) can be implemented by a functorial algebra with the *Maybe* functor as the functor carrier with the following *EndoAlg*:

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

For the base carrier interpreting *throw* and *catch* that are not enclosed by other *catch*, a straightforward choice is just taking *Maybe a* as the base carrier for any type *a*, and setting *callB* = *callE* and *enterB* = *enterE*, which means operations inside and outside scopes are interpreted in the same way. In general, we define a function that gets us a *BaseAlg* from *EndoAlg* for free:

$$\begin{array}{l} \mathit{freeB} :: \mathit{EndoAlg}\ \Sigma\ \Gamma\ f \to \mathit{BaseAlg}\ \Sigma\ \Gamma\ f\ (f\ a) \\ \mathit{freeB}\ (\mathit{EndoAlg}\ \{..\}) = \mathit{BaseAlg}\ \mathit{callE}\ \mathit{enterE} \end{array} \tag{8}$$

The handler can be applied to programs using the *handle* function in Figure 1. For example, running

$$\mathit{handle}\ \mathit{excE}\ (\mathit{freeB}\ \mathit{excE})\ \mathit{Just}\ (\mathbf{do}\ \{x \leftarrow \mathit{catch}\ \mathit{throw}\ (\mathit{return}\ 42); \mathit{return}\ (x+1)\}) \tag{9}$$

produces *Just* 42 as expected. For the non-standard semantics (cf. handler (2)) that exception recovery is disabled, one can define *excE′* with the same *callE* as *exeE* but with

```
enterE′ :: Catch (Maybe (Maybe a)) → Maybe a
enterE′ (Catch Nothing _) = Nothing            enterE′ (Catch (Just k) _) = k
```

With *excE′*, handling the program in (9) produces *Nothing* as expected.

Now we provide some intuition for how functorial algebras work. First note that the three fields of *EndoAlg* in Figure 1 precisely correspond to the three cases of *Prog* (6). Thus by replacing the constructors of *Prog* with the correspond fields of *EndoAlg*, we have a polymorphic function $\mathit{hcata}\ \mathit{ealg} :: \forall x.\ \mathit{Prog}\ \Sigma\ \Gamma\ x \to f\ x$ (Figure 1) turning a program into a value in f.

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

We close this section with one more example of handling nondeterminism with a semi-determinism operator *once*. More interesting examples can be found in the appendix both in Haskell and OCaml, for example, handling parallel composition of current processes with the resumption monad.

Example 2.4. The standard way to handle explicit nondeterminism (Example 2.2) is by a functorial algebra with the list functor [ ] as the functor carrier together with the following algebra:

```
ndetE :: EndoAlg Choice Once [ ]
ndetE = EndoAlg {..} where          enterE :: Once [[a]] → [a]
   callE :: Choice [a] → [a]        enterE (Once x) = if x ≡ [ ] then [ ] else head x
   callE Fail      = [ ]            returnE :: a → [a]
   callE (Or x y) = x ++ y          returnE x = [x]
```

Then handling the following program with *ndetE* and *freeB ndetE* produces [1, 2] as expected.

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

In comparison, if *once* were algebraic, the result would be [1].

3 CATEGORICAL FOUNDATIONS FOR SCOPED OPERATIONS

Having seen programming examples of scoped operations with functorial algebras in Haskell, we now move on to the principles of programming languages with scoped operations—a categorical foundation for scoped operations and functorial algebras. First, we recall some standard category theory underlying algebraic effects and handlers (Section 3.1) and also Piróg et al. [2018]'s monad P that models the syntax of scoped operations, which is exactly the *Prog* monad in the Haskell implementation (Section 3.2). Then, we define functorial algebras formally (Section 3.3) and show that there is an adjunction between the category of functorial algebras and the base category (Section 3.4). Crucially, this adjunction induces a monad T isomorphic to Piróg et al. [2018]'s P, and thus functorial algebras can be used to interpret the syntax of scoped operations.

Notation. We assume familiarity with basic category theory, such as adjunctions, monads, and initial algebras, which are covered by standard texts [Barr and Wells 1990; Mac Lane 1998]. For notation, we write $f \cdot g$ for vertical composition of morphisms, and $F \circ G$ for horizontal composition of natural transformations and composition of functors. Products in a category are denoted by $X_1 \times \cdots \times X_n$ for finite products or $\prod_{i \in I} X_i$ for some set I, and coproducts are denoted by $X_1 + \cdots + X_n$ or $\coprod_{i \in I} X_i$ for some set I. The injection morphisms into coproducts are denoted by $\iota_i : X_i \to X_1 + \cdots + X_n$ for each i, and the morphisms out of coproducts are $[f_1, \ldots, f_n] : X_1 + \cdots + X_n \to Z$ for objects Z with morphisms $f_i : X_i \to Z$ for all i. Lastly, the (carrier of the) *initial algebra* of an endofunctor $G : \mathbb{C} \to \mathbb{C}$ is denoted by μG or $\mu Y.\, GY$, and the unique morphism from the initial G-algebra to a G-algebra $\alpha : GX \to X$, i.e. the catamorphism for α, is denoted by $(\!|\alpha|\!) : \mu G \to X$.

3.1 Syntax and Semantics of Algebraic Operations

The relationships between *equational theories*, *Lawvere theories*, *monads*, and *computational effects* are well-studied for decades from many perspectives [Hyland and Power 2007; Kelly and Power 1993; Moggi 1991; Plotkin and Power 2002; Power 1999; Robinson 2002]. Here we recap a simplified version of equational theories by Kelly and Power [1993] that we use to model the syntax and semantics of algebraic operations.

Algebraic Operations on Sets. In its simplest form, a *signature* Σ of operations consists of a set of *operation symbols* $\{ o_i \}_{i \in I}$ and a mapping from each operation symbol o_i to a natural number $a(o_i)$ called the *arity* of the operation. A Σ*-algebra* or a Σ*-model* is a set X, called the *carrier*, and a family of functions $\alpha_{o_i} : X^{a(o_i)} \to X$ for each operation symbol o_i, called the *structure maps*. A Σ-algebra *homomorphism* from $\langle X, \alpha \rangle$ to $\langle X', \alpha' \rangle$ is a function $h : X \to X'$ such that for all operation symbols o_i and all $\langle x_1, \ldots, x_{a(o_i)} \rangle \in X^{a(o_i)}$,

$$h(\alpha_{o_i}(x_1, \ldots, x_{a(o_i)})) = \alpha'_{o_i}(h(x_1), \ldots, h(x_{a(o_i)}))$$

Given any set Y, the set $\Sigma^* Y$ of *terms t generated by* Y is inductively given by two rules:

$$\frac{y \in Y}{y \in \Sigma^* Y} \qquad\qquad \frac{a(o_i) = n \qquad t_1, \ldots, t_n \in \Sigma^* Y}{o_i(t_1, \ldots, t_n) \in \Sigma^* Y}$$

The set Y can be intuitively understood as a set of *variables* in terms. For each operation o_i, there is a function $\mathrm{op}_{o_i} : (\Sigma^* Y)^{a(o_i)} \to \Sigma^* Y$ building a term from subterms and operation o_i, so $\langle \Sigma^* Y, \mathrm{op} \rangle$ is a Σ-algebra, called the *free* Σ*-algebra* generated by Y. The idea is that terms model the syntax of operations and algebras give semantics to operations, and thus for any algebra $\langle X, \alpha \rangle$ of Σ, every function $f : Y \to X$ induces an *interpretation* function $\mathit{handle}_{\langle X, \alpha \rangle}(f) : \Sigma^* Y \to X$ given by

$$y \mapsto f y \qquad\qquad o_i(t_1, \ldots, t_n) \mapsto \alpha_{o_i}(\mathit{handle}_{\langle X,\alpha \rangle}(f)(t_1), \ldots, \mathit{handle}_{\langle X,\alpha \rangle}(f)(t_n))$$

which is in fact a Σ-algebra homomorphism.

These constructions can be conveniently described in terms of category theory: a signature Σ is an endofunctor on the category $\texttt{Set}$ of sets in the form of $\Sigma = \coprod_{o_i} (-)^{a(o_i)}$, and Σ^* is an endofunctor $\texttt{Set} \to \texttt{Set}$. An algebra of Σ is a set X paired with a morphism $\Sigma X \to X$, and they form a category $\Sigma\texttt{-Alg}$ whose morphisms are algebra homomorphisms. The interpretation $\mathit{handle}_{\langle X,\alpha\rangle}$ is a function $\texttt{Set}(Y, X) \to \texttt{Set}(\Sigma^* Y, X)$, and the image of the function is in $\Sigma\texttt{-Alg}(\langle \Sigma^* Y, \mathrm{op}\rangle, \langle X, \alpha\rangle)$.

Locally Finitely Presentable Categories. These concepts are not tied to the category of sets or only operations with finite arity, and many generalisations have been considered in the literature. In this paper, we follow a common generalisation (e.g., as in [Ghani et al. 2003, 2006; Kelly and Power 1993; Power 1999]) that abstracts away from the category of sets to any *locally finitely presentable* (lfp) categories [Adamek and Rosicky 1994], which is useful for applications in programming languages, but unlike Kelly and Power [1993], we only consider *unenriched* categories in this paper.

The use of lfp categories in this paper is limited to some standard results about the existence of many initial algebras in lfp categories, and thus a reader not familiar with lfp categories may follow this paper with some simple intuition: a category $\mathbb{C}$ is lfp if it has all (small) colimits and a (small) set of *finitely presentable* objects in $\mathbb{C}$ such that every object in $\mathbb{C}$ can be obtained by 'glueing' (formally called *filtered colimits*) some finitely presentable objects. For example, $\texttt{Set}$ is an lfp category with finite sets as its finitely presentable objects, and indeed every set can be obtained by glueing, here meaning taking the union of, all its finite subsets:

$$X = \bigcup \{\, N \subseteq X \mid N \text{ finite} \,\}$$

Other examples of lfp categories include the category of partially ordered sets, the category of graphs, the category of small categories, and presheaf categories, thus lfp categories are widespread to cover many semantic settings of programming languages. Moreover, an endofunctor $F : \mathbb{C} \to \mathbb{C}$ is said to be *finitary* if it preserves 'glueing' (filtered colimits), which implies that its values FX are determined by its values at finitely presentable objects:

$$FX = F(\texttt{colim}_i N_i) \cong \texttt{colim}_i F N_i$$

where N_i are the finitely presentable objects that generate X when glued together. For example, polynomial functors $\coprod_{n \in \mathbb{N}} Pn \times (-)^n$ on $\texttt{Set}$ are finitary where Pn is a set for every n, but the endofunctor $(-)^S$ is not finitary when S is an infinite set.

Algebraic Operations on LFP Categories. Now we generalise operations and terms on $\texttt{Set}$ to on lfp categories. Given any lfp category $\mathbb{C}$, we simply take finitary endofunctors $\Sigma : \mathbb{C} \to \mathbb{C}$ as signatures of operations on $\mathbb{C}$. This is compatible with our earlier definition of signatures on sets because when $\mathbb{C} = \texttt{Set}$, polynomial functors $\coprod_{o_i} (-)^{a(o_i)}$ are finitary. The category $\Sigma\texttt{-Alg}$ of Σ-algebras is defined as usual: it has pairs $\langle X : \mathbb{C}, \alpha : \Sigma X \to X\rangle$ as objects and morphisms $h : X \to X'$ such that $h \cdot \alpha = \alpha' \cdot \Sigma h$ as morphisms $\langle X, \alpha\rangle \to \langle X', \alpha'\rangle$. The following classical results (see e.g. [Adámek 1974; Barr 1970]) give sufficient conditions for constructing initial and free Σ-algebras:

Lemma 3.1. *If category $\mathbb{C}$ has an initial object and colimits of all ω-chains and endofunctor $\Sigma : \mathbb{C} \to \mathbb{C}$ preserves them, then there exists an initial Σ-algebra $\langle \mu\Sigma : \mathbb{C},\ \mathit{in} : \Sigma(\mu\Sigma) \to \mu\Sigma\rangle$. Moreover, the structure map in is an isomorphism.*

Lemma 3.2. *If category $\mathbb{C}$ has finite coproducts and colimits of all ω-chains and endofunctor Σ : $\mathbb{C} \to \mathbb{C}$ preserves them, then the forgetful functor $\Sigma\texttt{-Alg} \to \mathbb{C}$ has a left adjoint $\texttt{Free}_\Sigma : \mathbb{C} \to \Sigma\texttt{-Alg}$, which maps every object X to $\mu Y.\, X + \Sigma Y$ with structure map*

$$op_X = \left(\Sigma(\mu Y.\, X + \Sigma Y) \xrightarrow{\iota_2} X + \Sigma(\mu Y.\, X + \Sigma Y) \xrightarrow{in} \mu Y.\, X + \Sigma Y\right)$$

and the unit η and counit ϵ of the adjunction $\mathtt{Free}_\Sigma \dashv \mathsf{U}_\Sigma$ are

$$\eta_X = in \cdot \iota_1 : X \longrightarrow U_\Sigma(\mathtt{Free}_\Sigma X) \qquad \epsilon_{\langle C, \beta : FC \to C\rangle} = (\![[id, \beta]]\!) : \mathtt{Free}_\Sigma C \longrightarrow \langle C, \beta\rangle \tag{10}$$

where $(\![[id, \beta]]\!)$ denotes the catamorphism from the initial algebra $\mu Y.\ C + \Sigma Y$ to C. Moreover, this adjunction is strictly monadic.

Lemma 3.2 is applicable to our setting since $\mathbb{C}$ being lfp directly implies that it has all colimits, and finitary functors Σ preserve colimits of ω-chains because colimits of ω-chains are filtered colimits. Hence we have the following adjunction: $\mathtt{Free}_\Sigma \dashv \mathsf{U}_\Sigma : \Sigma\text{-}\mathtt{Alg} \to \mathbb{C}$. We denote the monad from the adjunction by $\Sigma^* = \mathsf{U}_\Sigma \mathtt{Free}_\Sigma$. The idea is still that syntactic terms built from operations in Σ are modelled by the monad Σ^*, and semantics of operations are given by Σ-algebras. Given any Σ-algebra $\langle X, \alpha : \Sigma X \to X\rangle$ and morphism $g : A \to X$ in $\mathbb{C}$, the evaluation morphism $\Sigma^* A \to X$ is

$$handle_{\langle X,\alpha\rangle} g = \mathsf{U}_\Sigma(\epsilon_{\langle X,\alpha\rangle} \cdot \mathtt{Free}_\Sigma g) : \Sigma^* X = \mathsf{U}_\Sigma \mathtt{Free}_\Sigma A \longrightarrow X$$

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

The perspective of Plotkin and Power [2002] is that computational effects are modelled by signatures Σ of primitive effectful operations, and they determine monads Σ^* that represent programs: units $\eta : A \to \Sigma^* A$ represent computations causing no effect, and multiplications $\mu : \Sigma^*\Sigma^* X \to \Sigma^* X$ mean *sequential composition* of programs. Additionally, Σ-algebras are *handlers* [Plotkin and Pretnar 2013] of operations that can be applied to programs to interpret the operations in a program.

The approach of algebraic effects has led to a significant body of research on programming with effects and handlers due to the inherent modularity in this approach, but it imposes an assumption on the operations to be modelled: by the construction of Σ^*, the multiplication of the monad Σ^* satisfies the *algebraicity* property: $\mathrm{op} \cdot (\Sigma \circ \mu) = \mu \cdot (\mathrm{op} \circ \Sigma^*)$, where $\mathrm{op} : \Sigma(\Sigma^*) \to \Sigma^*$ is the structure map of free algebras. This intuitively means that every operation in Σ must be commutative with sequential composition of computations. Many, but not all, effectful operations satisfy this property, and they are called *algebraic operations*.

The crux of what we have seen is the adjunction $\mathtt{Free}_\Sigma \dashv \mathsf{U}_\Sigma$: syntax is modelled by the monad on the base category $\mathbb{C}$ arising from the adjunction, and semantics is given by the objects on the other side of the adjunction. We do not even need the adjunction to be the free/forgetful adjunction. For any monad $P : \mathbb{C} \to \mathbb{C}$ that models programs with some computational effects, if $L \dashv R : \mathbb{D} \to \mathbb{C}$ is an adjunction such that $RL \cong P$ as monads, then objects D in $\mathbb{D}$ provide a means to interpret programs PA: for any $g : A \to RD$ in $\mathbb{C}$, we have the following interpretation morphism

$$handle_D g = R(\epsilon_D \cdot Lg) : PA \cong R(LA) \longrightarrow RD \tag{11}$$

The intuition for g is that it transforms the returned value A of a computation into the carrier RD, so it corresponds to the 'value case' of effect handlers [Bauer and Pretnar 2015]. Piróg et al. [2018] call this approach the *adjoint-theoretic approach to syntax and semantics*, and they construct an adjunction between *indexed algebras* and the base category for modelling scoped operations. Before this, Levy [2003a] and Kammar and Plotkin [2012] also adopt a similar adjunction-based viewpoint in the treatment of call-by-push-value calculi: types representing *pure values* are interpreted in the base category $\mathbb{C}$, and types representing *computations* are interpreted in the category $\mathbb{D}$.

Remark 3.1. A notable missing part of our treatment is the *equations* that specify operations in a signature. Following Kelly and Power [1993], an equation for a signature $\Sigma : \mathbb{C} \to \mathbb{C}$ can be formulated as a pair of monad morphisms $\sigma, \tau : \Gamma^* \to \Sigma^*$ for some finitary functor Γ, and taking their coequaliser $\Gamma^* \overset{\tau}{\underset{\sigma}{\rightrightarrows}} \Sigma^* \twoheadrightarrow M$ in the category of finitary monads constructs a monad M that represents terms modulo the equation $l = r$. Although it seems possible to extend this formulation to work with scoped operations in theory, equations are hard to be checked in implementations. Thus, for the sake of simplicity, we do not consider equations in this paper.

Remark 3.2. Working with lfp categories and finitary functors precludes operations with infinite arguments, such as the *get* operation of mutable state when the state has infinite possible values, but this limitation is not inherent and can be handled by moving to *locally κ-presentable categories* for some larger cardinal κ [Ghani et al. 2003].

3.2 Syntax of Scoped Operations

Not all operations in programming languages can be adequately modelled as algebraic operations on Set, for example, λ-abstraction [Fiore et al. 1999], memory cell generation [Levy 2003b; Plotkin and Power 2002], more generally, effects with dynamically generated instances [Staton 2013], explicit substitution [Ghani et al. 2006], channel restriction in π-calculus [Stark 2008], and their syntax need to be modelled in some functor categories. More recently, Piróg et al. [2018] extend Ghani et al. [2006]'s work to model the syntax of a family of non-algebraic operations, which they call *scoped operations*, as an initial algebra μG in an endofunctor category $\mathbb{C}^{\mathbb{C}}$ for some higher-order endofunctor $G : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}^{\mathbb{C}}$. Moreover, the initial algebra $\mu G : \mathbb{C} \to \mathbb{C}$ can be given a monad structure. In this subsection, we review their development in the setting of lfp categories.

We fix an lfp category $\mathbb{C}$ in the rest of this paper, and refer to it as the *base category*, and it is intended to be the category in which types of a programming language are interpreted. Furthermore, we fix two finitary endofunctors $\Sigma, \Gamma : \mathbb{C} \to \mathbb{C}$ and call them the signature of algebraic operations and scoped operations respectively.

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

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

where $o \in \Sigma n$ is an algebraic operation of n arguments, and similarly $s \in \Gamma n$ is a scoped operation that creates n scopes. Note that unlike the second rule for algebraic operations, the third rule for scoped operations also takes a continuation $k : X \to PA$, as it is *not* necessarily the case that $\{s(p) \gg\!= k\} = s(k^\dagger \cdot p)$ where $k^\dagger : PX \to PA$ is the Kleisli extension of k. When $\mathbb{C}$ is any lfp category, these rules translate to the following recursive equation for the monad $P : \mathbb{C} \to \mathbb{C}$:

$$PA \cong A + \Sigma(PA) + \int^{X:\mathbb{C}} \coprod_{\mathbb{C}(X,PA)} \Gamma(PX) \tag{12}$$

where $\int^{X:\mathbb{C}}$ denotes a *coend* [Mac Lane 1998, Chapter IX] in $\mathbb{C}$, which is analogous to existential types in polymorphic λ-calculus. Moreover, the coend in (12) is isomorphic to $\Gamma(P(PA))$ because it exactly computes $\mathsf{Lan}_I(\Gamma P)(PA)$, i.e. the value of the left Kan-extension of ΓP along the identity functor $I : \mathbb{C} \to \mathbb{C}$ at PA, and clearly $\mathsf{Lan}_I(\Gamma P) = \Gamma P$. Thus (12) is equivalent to

$$PA \cong A + \Sigma(PA) + \Gamma(P(PX)) \tag{13}$$

which is exactly the *Prog* Σ Γ datatype that we saw in the Haskell implementation (6). To obtain a solution to (13), we construct a (higher-order) endofunctor $G : \mathsf{Endo}_f(\mathbb{C}) \to \mathsf{Endo}_f(\mathbb{C})$ to represent the Grammar where $\mathsf{Endo}_f(\mathbb{C})$ is the category of finitary endofunctors on $\mathbb{C}$:

$$G = Id + \Sigma \circ - + \Gamma \circ - \circ - \tag{14}$$

where $Id : \mathbb{C} \to \mathbb{C}$ is the identity functor. Then Lemma 3.2 is applicable because $\mathsf{Endo}_f(\mathbb{C})$ has all small colimits since colimits in functor categories can be computed pointwise and $\mathbb{C}$ has all small colimits. Furthermore, G preserves all filtered colimits, in particular colimits of ω-chains, because $- \circ = : \mathsf{Endo}_f(\mathbb{C}) \times \mathsf{Endo}_f(\mathbb{C}) \to \mathsf{Endo}_f(\mathbb{C})$ is finitary following from direct verification. By Lemma 3.1, there is an initial G-algebra $\langle P : \mathsf{Endo}_f(\mathbb{C}), \mathsf{in} : GP \to P\rangle$ and in is an isomorphism. Thus P obtained in this way is indeed a solution to (13)—the endofunctor modelling the syntax of programs with algebraic and scoped operations.

Monadic Structure of P. Next we equip the finitary endofunctor P with a monad structure. This can be done in several ways, either by the general result about Σ-*monoids* [Fiore and Hur 2009a; Fiore et al. 1999] in $\mathsf{Endo}_f(\mathbb{C})$, or by [Matthes and Uustalu 2004, Theorem 4.3], or by the following relatively straightforward argument in [Piróg et al. 2018]: by the 'diagonal rule' of computing initial algebras by Backhouse et al. [1995], $P = \mu G$ (14) is isomorphic to $P' = \mu X.\ Id + \Sigma \circ X + \Gamma \circ P \circ X$. Note that P' is exactly $(\Sigma + \Gamma \circ P)^*$ as endofunctors by Lemma 3.2, thus

$$P \cong (\Sigma + \Gamma \circ P)^* : \mathsf{Endo}_f(\mathbb{C}) \tag{15}$$

Then we equip P with the same monad structure as $(\Sigma + \Gamma \circ P)^*$. The implementation in (6) is exactly this monad structure.

3.3 Functorial Algebras of Scoped Operations

To interpret the monad $P = \mu G$ (14) modelling the syntax of scoped operations, it is natural to expect that semantics is given by G-algebras on $\mathsf{Endo}_f(\mathbb{C})$ so that interpretation is then the catamorphisms from μG to G-algebras. And following the adjoint-theoretic approach (11), we would like to have an adjunction $\begin{tikzcd} G\text{-}\mathtt{Alg} \arrow[r, shift right] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right] \end{tikzcd}$ such that the induced monad is isomorphic to P. However, there seems no natural way to construct such an adjunction unless we replace G-algebras with a slight extension of it, which we referred to as *functorial algebras*, as the notion for giving semantics to scoped operations. In the following, we first define functorial algebras formally (Definition 3.1) and then show the adjunction between the category of functorial algebras and the base category (Theorem 3.6), which allows us to interpret P with functorial algebras.

A functorial algebra is carried by a finitary endofunctor $H : \mathbb{C} \to \mathbb{C}$ with additionally an object X in $\mathbb{C}$. The endofunctor H also comes with a morphism $\alpha^G : GH \to H$ in $\mathsf{Endo}_f(\mathbb{C})$, and the object X is equipped with a morphism $\alpha^I : \Sigma X + \Gamma H X \to X$ in $\mathbb{C}$. The intuition is that given a program of type $PX \cong X + \Sigma(PX) + \Gamma(P(PX))$, the middle P in ΓPP corresponds to the part of a program enclosed by some scoped operations (i.e. the p in $\{s(p) \gg\!= k\}$), and this part of the program is interpreted by H with α^G. After the enclosed part is interpreted, α^I interprets the outermost layer of the program by X with α^I in the same way as interpreting free monads of algebraic operations. More precisely, let $I : \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathbb{C}$ be a bi-functor such that [1]

$$I_H X = \Sigma X + \Gamma(HX) \qquad I_\sigma f = \Sigma f + \Gamma(\sigma \circ f) \tag{16}$$

where $\circ$ is horizontal composition, for all $H : \mathsf{Endo}_f(\mathbb{C})$ and $X : \mathbb{C}$ and all morphisms $\sigma : H \to H'$ and $f : X \to X'$. Then we define an endofunctor

$$\mathsf{Fn} : \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \qquad \text{such that} \qquad \mathsf{Fn}\langle H, X\rangle = \langle GH, I_H X\rangle \tag{17}$$

with the evident action on morphisms. We call Fn-algebras *functor algebras*.

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

Definition 3.1 (Functorial Algebras). A *functorial algebra* is an object $\langle H, X\rangle$ in $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ paired with a structure map $\mathsf{Fn}\langle H, X\rangle \longrightarrow \langle H, X\rangle$, or equivalently it is a quadruple

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

where $GH = Id + \Sigma \circ H + \Gamma \circ H \circ H$. Morphisms between two functorial algebras $\langle H_1, X_1, \alpha^G_1, \alpha^I_1\rangle$ and $\langle H_2, X_2, \alpha^G_2, \alpha^I_2\rangle$ are pairs $\langle \sigma : H_1 \longrightarrow H_2,\ f : X_1 \longrightarrow X_2\rangle$ making the following diagrams commute:

$$\begin{tikzcd} GH_1 \arrow[r, "\alpha^G_1"] \arrow[d, "G\sigma"'] & H_1 \arrow[d, "\sigma"] \\ GH_2 \arrow[r, "\alpha^G_2"'] & H_2 \end{tikzcd} \qquad \begin{tikzcd} \Sigma X_1 + \Gamma(H_1 X_1) \arrow[r, "\alpha^I_1"] \arrow[d, "\Sigma f + \Gamma(\sigma \circ f)"'] & X_1 \arrow[d, "f"] \\ \Sigma X_2 + \Gamma(H_2 X_2) \arrow[r, "\alpha^I_2"'] & X_2 \end{tikzcd}$$

Functorial algebras and their morphisms form a category $\mathsf{Fn\text{-}Alg}$.

Example 3.1. We reformulate our programming example of nondeterministic choice with *once* shown Example 2.4 in the formal definition. Let $\mathbb{C} = \mathsf{Set}$ in this example and $1 : \mathsf{Set}$ be some singleton set. We define signature endofunctors

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

so that Σ represents nullary algebraic operation *fail* and binary algebraic operation *or*, and Γ represents the unary scoped operation *once* that creates one scope. Let $List : \mathsf{Set} \longrightarrow \mathsf{Set}$ be the endofunctor mapping a set X to the set of (finite) lists with elements from X. We define natural transformations $\alpha^\Sigma : \Sigma \circ List \longrightarrow List$ and $\alpha^\Gamma : \Gamma \circ List \circ List \longrightarrow List$ by

$$\alpha^\Sigma_X(\iota_1\ \star) = nil \qquad \alpha^\Sigma_X(\iota_2\ \langle x, y\rangle) = x +\!\!+ y \qquad \alpha^\Gamma_X(nil) = nil \qquad \alpha^\Gamma_X(cons\ x\ xs) = x$$

where $\star$ is the only element in singleton set 1; *nil* is the empty list; $+\!\!+$ is list concatenation; and *cons x xs* is the list with an element x in front of *xs*. Then for any set X, $\langle List, List\ X\rangle$ carries a functorial algebra with structure maps

$$\alpha^G = [\eta^{List}, \alpha^\Sigma, \alpha^\Gamma] : GList \longrightarrow List \qquad\qquad \alpha^I = [\alpha^\Sigma_X, \alpha^\Gamma_X] : I_{List}X \longrightarrow X \tag{18}$$

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

The last example demonstrates that one can define functorial algebras from G-algebras (14) by simply letting the object component X to be HX where H is the endofunctor component and X is an arbitrary object in $\mathbb{C}$. This practically means that the outermost layer of a program—by which we mean the part of a program that is not enclosed in any scoped operations—and the inner layers are interpreted in the same way.

Proposition 3.3. *For the functor G in (14) and every $X : \mathbb{C}$, there is a faithful functor $J_X : G\text{-}\mathsf{Alg} \longrightarrow$ $\mathsf{Fn\text{-}Alg}$ that maps any G-algebra $\langle H, \alpha : GH \longrightarrow H\rangle$ to the functorial algebra $\langle H, HX, \alpha, \beta\rangle$ where*

$$\beta = [\alpha_X \cdot \iota_2, \alpha_X \cdot \iota_3] : \Sigma(HX) + \Gamma(H(HX)) \longrightarrow HX$$

Example 3.2. Compared to G-algebras, the object component of functorial algebras offers the flexibility that the outmost layer of a program can be interpreted differently from the inner layers. For example, to interpret nondeterministic choices and *once* as in Example 3.1, if one is only interested in the final number of possible outcomes, then one can define a functorial algebra $\langle List, \mathbb{N}, \alpha^G, \alpha^I\rangle$ where α^G is (18) and

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

3.4 Interpreting with Functorial Algebras

In the rest of this section we show how functorial algebras can be used to interpret programs PA (13) with scoped operations. We first construct a simple adjunction $\uparrow \dashv \downarrow$ between the base category $\mathbb{C}$ and $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$, which is then composed with the free/forgetful adjunction $\mathsf{Free}_{\mathsf{Fn}} \dashv \mathsf{U}_{\mathsf{Fn}}$ between $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ and $\mathsf{Fn\text{-}Alg}$ for the functor Fn (17). The resulting adjunction (19) is proven to induce a monad T isomorphic to P (Theorem 3.6), and by the adjoint-theoretic approach to syntax and semantics (11), this adjunction provides a means to interpret scoped operations modelled with the monad P (Theorem 3.7).

We start with constructing an adjunction $\uparrow \dashv \downarrow$ between $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ and $\mathbb{C}$. Define functor

$$\uparrow : \mathbb{C} \to \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \qquad \text{such that} \qquad \uparrow X = \langle 0, X \rangle$$

where $0 : \mathsf{Endo}_f(\mathbb{C})$ is the initial endofunctor—the constant functor sending everything to the initial object in $\mathbb{C}$. The functor $\uparrow$ is left adjoint to the projection functor $\downarrow : \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathbb{C}$ mapping $\langle H, X \rangle$ to X with the obvious action on morphisms.

Then we would like to compose $\uparrow \dashv \downarrow$ with the free-forgetful adjunction $\mathsf{Free}_{\mathsf{Fn}} \dashv \mathsf{U}_{\mathsf{Fn}}$ for the higher-order endofunctor Fn (17) on $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$. The latter indeed exists by the following lemma.

Lemma 3.4. *The endofunctor* Fn *(17) on* $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ *has free algebras, i.e. there is a functor* $\mathsf{Free}_{\mathsf{Fn}} : \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \to \mathsf{Fn\text{-}Alg}$ *left adjoint to the forgetful functor* $\mathsf{U}_{\mathsf{Fn}} : \mathsf{Fn\text{-}Alg} \to \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$.

Proof. Since $\mathbb{C}$ is lfp, it is cocomplete, and it follows that $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ is cocomplete because colimits in functor categories and product categories can be computed pointwise [Mac Lane 1998, Thm V.3.2]. It is also easy verification that Fn preserves all colimits of ω-chains following from the fact that G, Σ and Γ, and all functors in $\mathsf{Endo}_f(\mathbb{C})$ preserve colimits of ω-chains in their domains respectively. Hence by Lemma 3.2, there is a functor $\mathsf{Free}_{\mathsf{Fn}}$ left adjoint to U_{Fn}. □

The two adjunctions that we have just constructed are depicted in the following diagram:

$$\begin{tikzcd}
\mathsf{Fn\text{-}Alg} \arrow[r, bend right, "\mathsf{U}_{\mathsf{Fn}}"'] \arrow[r, phantom, "\perp"] & \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, bend right, "\mathsf{Free}_{\mathsf{Fn}}"'] \arrow[r, bend right, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\uparrow"'] \arrow[loop right, "T"]
\end{tikzcd} \tag{19}$$

and we compose them to obtain an adjunction $\mathsf{Free}_{\mathsf{Fn}} \uparrow \dashv \downarrow \mathsf{U}_{\mathsf{Fn}}$ between $\mathsf{Fn\text{-}Alg}$ and $\mathbb{C}$, giving rise to a monad $T = \downarrow \mathsf{U}_{\mathsf{Fn}} \mathsf{Free}_{\mathsf{Fn}} \uparrow$. In the rest of this section, we prove that T is isomorphic to P (12) in the category of monads, which is crucial in this paper, since it allows us to interpret scoped operations modelled by the monad P with functorial algebras $\mathsf{Fn\text{-}Alg}$.

We first establish the following lemma characterising the free Fn-algebra on the product category $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ in terms of the free algebras in $\mathbb{C}$ and $\mathsf{Endo}_f(\mathbb{C})$. The idea of this lemma is that since the first component of $\mathsf{Fn}\langle H, X \rangle$ (17) is GH, which does not depend on X, thus the first component of the carrier of the free Fn-algebra $\mathsf{Free}_{\mathsf{Fn}}\langle H, X \rangle$ should not depend on X either because colimits in product category $\mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C}$ can be calculated component-wise.

Lemma 3.5. *There is a natural isomorphism between* $\mathsf{Free}_{\mathsf{Fn}}$ *and the following functor*

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

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

Proof Sketch. It follows from the formula of computing free algebras from initial algebras Lemma 3.2. A detailed proof can be found in the appendix. □

Theorem 3.6. *The monad* P *is isomorphic to* T *in the category of monads.*

Proof. By (15) and (16), $P \cong (\Sigma + \Gamma P)^* = (I_P)^*$ as monads, it is sufficient to show that $T \cong (I_P)^*$ as monads. Recall that $P = \mu G \cong G^*0$ as endofunctors. Let $\psi : G^*0 \to P$ be the isomorphism, and let ϕ be the isomorphism between Fn^* and $\mathsf{U}_{\mathsf{Fn}}\widehat{\mathsf{Free}_{\mathsf{Fn}}}$ by Lemma 3.5, then for all $X : \mathbb{C}$, we have

$$TX = \downarrow(\mathsf{U}_{\mathsf{Fn}}(\mathsf{Free}_{\mathsf{Fn}}(\uparrow X))) = \downarrow(\mathsf{Fn}^*\langle 0, X\rangle) \overset{\downarrow\phi}{\cong} \downarrow\langle G^*0, \langle I_{G^*0}\rangle^* X\rangle = (I_{G^*0})^* X \overset{(I_\psi)^*}{\cong} (I_P)^* X \tag{20}$$

Thus T is isomorphic to $(I_P)^*$ as endofunctors. What remains is to show that this isomorphism preserves the units and multiplications of T and P. The proof for this is tedious calculation and not very instructive, thus omitted here. A detailed proof can be found in the appendix. □

Remark 3.3. In general, the adjunction $\mathsf{Free}_{\mathsf{Fn}}\uparrow \dashv \downarrow\mathsf{U}_{\mathsf{Fn}}$ 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 4, functorial algebras and Eilenberg-Moore algebras have the same expressive power for interpreting scoped operations in the base category.

The isomorphism established Theorem 3.6 enables us to interpret programs modelled by the monad P using functorial algebras following Equation 11: for any functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$ (Definition 3.1), and any morphism $g : A \to X$ in the base category $\mathbb{C}$, there is a morphism

$$\mathit{handle}_{\langle H,X,\alpha^G,\alpha^I\rangle}\ g = \downarrow\mathsf{U}_{\mathsf{Fn}}(\epsilon_{\langle H,X,\alpha^G,\alpha^I\rangle} \cdot \mathsf{Free}_{\mathsf{Fn}}\uparrow g) : TA \cong PA \to X \tag{21}$$

which interprets programs PA with the functorial algebra $\langle H, X, \alpha^G, \alpha^I\rangle$. Furthermore, we can derive the following recursive formula (22) for this interpretation morphism, which is exactly the Haskell implementation that we saw in Figure 1.

Theorem 3.7 (Interpreting with Functorial Algebras). *For any functorial algebra $\alpha = \langle H, X, \alpha^G, \alpha^I\rangle$ as in Definition 3.1, and any morphism $g : A \to X$ for some A in the base category $\mathbb{C}$, let $h = (\!|\alpha^G|\!) : P \to H$ be the catamorphism from the initial G-algebra P to the G-algebra $\alpha^G : GH \to H$. The interpretation of PA with this algebra α and g satisfies*

$$\mathit{handle}_\alpha\ g = [g,\quad \alpha^I_\Sigma \cdot \Sigma(\mathit{handle}_\alpha\ g),\quad \alpha^I_\Gamma \cdot \Gamma h_X \cdot \Gamma P(\mathit{handle}_\alpha\ g)] \cdot \mathit{in}_A^{-1}\ :\ PA \to X \tag{22}$$

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

Proof Sketch. It can be calculated by plugging the formula of ϵ for the adjunction $\downarrow\mathsf{U}_{\mathsf{Fn}} \dashv \mathsf{Free}_{\mathsf{Fn}}\uparrow$ (Lemma 3.2) into (21). □

To summarise, in this section we have developed a notion of functorial algebras that we use to handle scoped operations. The heart of the development is the adjunction (19) that induces a monad isomorphic to the monad P (13) that models the syntax of programs with scoped operations, following which we derive a recursive formula (22) that interprets programs with functor algebras. The formula is exactly the implementation in Figure 1: the datatype *EndoAlg* represents the α^G in (22); datatype *BaseAlg* corresponds to α^I; function *hcata* implements $(\!|\alpha^G|\!)$; and the three arguments of *handle* implement the three components in (22) respectively.

4 COMPARING THE MODELS OF SCOPED OPERATIONS

Functorial algebras are not the only option for interpreting scoped operations. In this section we compare functorial algebras with two other approaches, one being Piróg et al. [2018]'s *indexed algebras* and the other one being *Eilenberg-Moore* (EM) algebras of the monad P (13), which simulate scoped operations with algebraic operations. After a brief description of these two kinds of algebras (Section 4.1 and Section 4.2), we compare the show that their expressive power is in fact equivalent.

```
data EMAlg Σ Γ x = EM {callEM :: Σ x → x, enterEM :: Γ (Prog Σ Γ x) → x}
handle_EM :: (Functor Σ, Functor Γ) ⇒ (EMAlg Σ Γ x) → (a → x) → Prog Σ Γ a → x
handle_EM alg gen (Return x) = gen x
handle_EM alg gen (Call op)  = (callEM alg · fmap (handle_EM alg gen)) op
handle_EM alg gen (Enter op) = (enterEM alg · fmap (fmap (handle_EM alg gen))) op
```

Fig. 2. Haskell implementation of EM algebras of P based on Theorem 4.2

4.1 Interpreting Scoped Operations with Eilenberg-Moore Algebras

In standard algebraic effects, handlers are just Σ-algebras for some signature functor $\Sigma : \mathbb{C} \to \mathbb{C}$, and it is well known that the category Σ-$\mathtt{Alg}$ of Σ-algebras is equivalent to the category $\mathbb{C}^{\Sigma^*}$ of EM algebras of the monad Σ^*. Thus handlers of algebraic operations are exactly EM algebras of the monad Σ^* modelling the syntax of algebraic operations. This observation suggests that we may also use EM algebras of the monad P (13) as the notion of handlers for scoped operations.

Lemma 4.1. *EM algebras of P are equivalent to $(\Sigma + \Gamma \circ P)$-algebras. In other words, an EM algebra of P is equivalently a tuple*

$$\langle X : \mathbb{C},\ \alpha_\Sigma : \Sigma X \to X,\ \alpha_\Gamma : \Gamma(PX) \to X\rangle \tag{23}$$

Proof. Recall that the monad structure of P (15) is exactly the monad structure of the free monad $(\Sigma + \Gamma \circ P)^*$, and therefore they have the same EM algebras. Moreover, EM algebras of $(\Sigma + \Gamma \circ P)^*$ are equivalent to plain $(\Sigma + \Gamma \circ P)$-algebra by the monadicity of the free-forgetful adjunction. □

Thus we obtain a way of interpreting scoped operations based on the free-forgetful adjunction $\mathtt{Free}_{\Sigma+\Gamma\circ P} \dashv \mathsf{U}_{\Sigma+\Gamma\circ P}$: given an EM algebra of P in the form of (23), then for any $A : \mathbb{C}$ and morphism $g : A \to X$, the interpretation of PA by g and this EM algebra is

$$\mathit{handle}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g = \mathsf{U}_{\Sigma+\Gamma\circ P}(\epsilon_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle} \cdot \mathtt{Free}_{\Sigma+\Gamma\circ P}\ g) : PA \cong (\Sigma + \Gamma \circ P)^* A \to X \tag{24}$$

The formula (24) can be turned into a recursive form that suits implementation (Figure 2).

Theorem 4.2 (Interpreting with EM Algebras). *Given an Eilenberg-Moore algebra as in (23), for any morphism $g : A \to X$ in $\mathbb{C}$ for some A, the interpretation of PA with this algebra and g satisfies*

$$\mathit{handle}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g = [g,\ \ \alpha_\Sigma \cdot \Sigma(\mathit{handle}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g),\ \ \alpha_\Gamma \cdot \Gamma P(\mathit{handle}_{\langle X,\alpha_\Sigma,\alpha_\Gamma\rangle}\ g)] \cdot \mathit{in}^\circ_A \tag{25}$$

where $\mathit{in}^\circ : P \to \mathit{Id} + \Sigma \circ P + \Gamma \circ P \circ P$ is the isomorphism between P and GP (13).

Proof. It can be calculated by plugging ϵ for $\mathtt{Free}_{\Sigma+\Gamma\circ P} \dashv \mathsf{U}_{\Sigma+\Gamma\circ P}$ (Lemma 3.2) into (24). □

Example 4.1. With the implementation of EM algebras and their interpretation (Figure 2), the scoped operation *once* in Example 2.4 can be interpreted by the following EM algebra

```
onceAlgEM :: EMAlg Choice Once [a]      enterEM :: Once (Prog Choice Once [a]) → [a]
onceAlgEM = EM {..} where               enterEM (Once p) =
  callEM :: Choice [a] → [a]              case handle_EM onceAlgEM (λx → [x]) p of
  callEM Fail       = [ ]                   [ ]     → [ ]
  callEM (Or x y) = x ++ y                  (x: _) → x
```

Note that this EM algebra is defined with recursion: the part of syntax enclosed by the scope of *once* is interpreted by a recursive call to handle_{EM} with the algebra itself.

Interpreting scoped operation with EM algebras can be understood as simulating scoped operations with algebraic operations and general recursion: a signature (Σ, Γ) of algebraic-and-scoped operations is simulated by a signature $(\Sigma + \Gamma \circ P)$ of algebraic operations where P is recursively given by $(\Sigma + \Gamma \circ P)^*$. In this way, one can simulate scoped operation in languages implementing algebraic effects that allow signatures of operation to be recursive, such as [Bauer and Pretnar 2014; Hillerström and Lindley 2018; Leijen 2017], but not the original design by Plotkin and Pretnar [2013], which requires signatures of operations to mention only some *base types*.

The downside of this simulating approach is that the denotational semantics of the language becomes more complex and usually involves solving some domain-theoretic recursive equations, like in [Bauer and Pretnar 2014]. Moreover, this approach typically requires handlers to be defined with general recursion, like in Example 4.1, which obscures the inherent structure of scoped operations, making reasoning about handlers of scoped operations more difficult.

4.2 Indexed Algebras of Scoped Effects

Indexed algebras of scoped operations by Piróg et al. [2018] are yet another way of interpreting scoped operations. They are based on the following adjunction:

$$\begin{tikzcd} \mathtt{Ix\text{-}Alg} \arrow[r, bend right, "\mathtt{U_{Ix}}"'] \arrow[r, phantom, "\perp"] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, bend right, "\mathtt{Free_{Ix}}"'] \arrow[r, bend right, "\downharpoonright"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\upharpoonright"'] \end{tikzcd} \tag{26}$$

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:

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

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

$$\langle A : \mathbb{C}^{|\mathbb{N}|},\ \ a : \Sigma \circ A \to A,\ \ d : \Gamma(\triangleleft A) \to A,\ \ p : A \to \triangleleft A \rangle \tag{27}$$

The operational intuition for it is that the carrier A_i at level i interprets the part of syntax enclosed by i layers of scopes, and when interpreting a scoped operation $\Gamma(P(PX))$ at layer i, the part of syntax outside the scope is first interpreted, resulting in $\Gamma(PA_i)$, and then the indexed algebra provides a way p to *p*romote the carrier to the next level, resulting in $\Gamma(PA_{i+1})$. After the inner layer is also interpreted as ΓA_{i+1}, the indexed algebra provides a way d to *d*emote the carrier, producing A_i again. Additionally the morphism a interprets ordinary *a*lgebraic operations. This process is implemented as the *hfold* function in Figure 4. Note that the code requires GHC's `DataKinds` extension for type-level natural numbers.

Example 4.2. Continuing Example 3.1, we can define an indexed algebra nondeterministic choice with *once* (Example 2.2). 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 the following structure maps: for all $i \in \mathbb{N}$,

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

The adjunction $\mathtt{Free_{Ix}} \dashv \mathtt{U_{Ix}}$ in (26) is the free-forgetful adjunction for $\mathtt{Ix}$ on $\mathbb{C}^{|\mathbb{N}|}$. The other adjunction $\upharpoonright \dashv \downharpoonright$ is given by $\downharpoonright A = A_0$, $(\upharpoonright X)_0 = X$, and $(\upharpoonright X)_{i+1} = 0$ for all $i \in \mathbb{N}$. Importantly, Piróg

$$\begin{tikzcd}
\mathtt{Fn\text{-}Alg} \arrow[d, "K_{\mathtt{Ix}}^{\mathtt{Fn}}"] \arrow[r, phantom, "\perp"] & \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, bend right, "\mathtt{Free}_{\mathtt{Fn}}"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\uparrow"'] \arrow[d, equal] \\
\mathtt{Ix\text{-}Alg} \arrow[d, dashed, "K_{\mathtt{EM}}^{\mathtt{Ix}}"'] \arrow[r, phantom, "\perp"] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, bend right, "\mathtt{Free}_{\mathtt{Ix}}"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\upharpoonleft"'] \arrow[d, equal] \\
\mathbb{C}^P \arrow[uu, bend left=60, "K_{\mathtt{Fn}}^{\mathtt{EM}}"] \arrow[rr, phantom, "\perp"] & & \mathbb{C} \arrow[ll, bend right, "\mathtt{Free}^P"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathtt{Fn\text{-}Alg} \arrow[r, bend right, "\mathtt{U}_{\mathtt{Fn}}"'] & \mathsf{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[r, bend right, "\downarrow"'] & \mathbb{C} \\
\mathtt{Ix\text{-}Alg} \arrow[r, bend right, "\mathtt{U}_{\mathtt{Ix}}"'] & \mathbb{C}^{|\mathbb{N}|} \arrow[r, bend right, "\downharpoonleft"'] & \mathbb{C} \\
\mathbb{C}^P \arrow[rr, bend right, "\mathtt{U}^P"'] & & \mathbb{C}
\end{tikzcd}$$

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

et al. [2018] show that the monad induced by the adjunction (26) is isomorphic to monad P (13), thus indexed algebras can also be used to interpret scoped operations

$$\mathit{handle}_{\langle A,a,d,p\rangle}\ g = \downharpoonleft \mathtt{U}_{\mathtt{Ix}}(\epsilon_{\langle A,a,d,p\rangle} \cdot \mathtt{Free}_{\mathtt{Ix}} \upharpoonleft g) : PX \cong (\downharpoonleft \mathtt{U}_{\mathtt{Ix}}\mathtt{Free}_{\mathtt{Ix}} \upharpoonleft)X \to A_0 \tag{28}$$

in the same way as what we do for functorial algebras in Section 3.4. However, Piróg et al. [2018]'s *hfold* for interpreting with indexed algebras (Figure 4) is not a direct implementation of *handle*, and it is not obvious whether they actually coincide—this will be the subject of Section 5.2.

4.3 Comparison of Resolutions

Now we move on to the real subject of this section—comparing the expressivity of the three ways for interpreting scoped operations. Specifically, we construct *comparison functors* between the respective categories of the three kinds of algebras, which translate one kind of algebras to another in a way *preserving the induced interpretation* in the base category. Categorically, the three kinds of algebras correspond to three *resolutions* of the monad P, which form a category of resolutions (Definition 4.1) with comparison functors as morphisms. In this category, the Eilenberg-Moore resolution is the terminal object, and thus it automatically gives us comparison functors translating other kinds of algebras to EM algebras (Section 4.4). To complete the circle of translations, we then construct comparison functors $K_{\mathtt{Fn}}^{\mathtt{EM}} : \mathbb{C}^P \to \mathtt{Fn\text{-}Alg}$ translating EM algebras to functorial ones (Section 4.5) and $K_{\mathtt{Ix}}^{\mathtt{Fn}} : \mathtt{Fn\text{-}Alg} \to \mathtt{Ix\text{-}Alg}$ translating functorial algebras to indexed ones (Section 4.6). The situation is pictured in Figure 3.

Definition 4.1 (Resolutions and Comparison Functors [Lambek and Scott 1986])**.** Given a monad M on $\mathbb{C}$, the category $\mathsf{Res}(M)$ of *resolutions* of M has as objects adjunctions $\langle \mathbb{D}, L \dashv R : \mathbb{D} \to \mathbb{C}, \eta, \epsilon\rangle$ such that the induced monad RL is M. A morphism from a resolution $\langle \mathbb{D}, L \dashv R, \eta, \epsilon\rangle$ to $\langle \mathbb{D}', L' \dashv R', \eta', \epsilon'\rangle$ is a functor $K : \mathbb{D} \to \mathbb{D}'$, called a *comparison functor*, such that it commutes with the left and right adjoints, i.e. $KL = L'$ and $R'K = R$.

As summarised in Figure 3, we have seen adjunctions for indexed algebras, EM algebras and functorial algebras respectively, each inducing the monad P up to isomorphism, so each of them can be identified with an object in the category $\mathsf{Res}(E)$. For each resolution $\langle \mathbb{D}, L, R, \eta, \epsilon\rangle$, we have been using the objects D in $\mathbb{D}$ to interpret scoped operations modelled by P: for any morphism $g : A \to RD$ in $\mathbb{C}$, the interpretation of PA by D and g is $\mathit{handle}_D\ g = R(\epsilon_D \cdot Lg) : PA = RLA \to RD$. Crucially, we show that interpretations are preserved by comparison functors.

Lemma 4.3 (Preservation of Interpretation)**.** *Let $K : \mathbb{D} \to \mathbb{D}'$ be any comparison functor between resolutions $\langle \mathbb{D}, L, R, \eta, \epsilon\rangle$ and $\langle \mathbb{D}', L', R', \eta', \epsilon'\rangle$ of some monad $M : \mathbb{C} \to \mathbb{C}$. For any object D in $\mathbb{D}$ and any morphism $g : A \to RD$ in $\mathbb{C}$, it holds that*

$$\mathit{handle}_D\ g = \mathit{handle}_{KD}\ g : MA \to RD (= R'KD) \tag{29}$$

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

Proof. Since $L \dashv R$ 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*. Then by Proposition 1 in [Mac Lane 1998, page 99], it holds that $K\epsilon = \epsilon' K$, and we have

$$handle_{KD}\ g = R'(\epsilon'_{KD} \cdot L'g) = R'(K\epsilon_D \cdot L'g) = R\epsilon_D \cdot R'L'g = R\epsilon_D \cdot RLg = handle_D\ g \qquad \square$$

The implication of this lemma is that if there is a comparison functor K from some resolution $L \dashv R : \mathbb{D} \longrightarrow \mathbb{C}$ to $L' \dashv R' : \mathbb{D}' \longrightarrow \mathbb{C}$ of the monad P, then K can *translate* a $\mathbb{D}$ object to a $\mathbb{D}'$ object that preserves the induced interpretation. Thus the expressive power of $\mathbb{D}$ for interpreting P is not greater than $\mathbb{D}'$, in the sense that every $handle_D\ g$ 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.

4.4 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 $\mathsf{Res}(M)$ for any monad M, which means that there *uniquely exists* a comparison functor from *every* resolution to the Eilenberg-Moore resolution. Specifically, given a resolution $\langle \mathbb{D}, L, R, \eta, \epsilon \rangle$ of a monad M, the unique comparison functor K from $\mathbb{D}$ to the category $\mathbb{C}^M$ of the Eilenberg-Moore algebras is

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

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

Lemma 4.4. *There uniquely exist comparison functors $K^{\mathsf{Ix}}_{\mathsf{EM}} : \mathsf{Ix\text{-}Alg} \longrightarrow \mathbb{C}^P$ and $K^{\mathsf{Fn}}_{\mathsf{EM}} : \mathsf{Fn\text{-}Alg} \longrightarrow \mathbb{C}^P$ from the resolutions of indexed algebras and functorial algebras to the resolution of EM algebras.*

4.5 Translating EM Algebras to Functorial Algebras

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

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

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

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

Lemma 4.5. *Functor $K^{\mathsf{EM}}_{\mathsf{Fn}}$ is a comparison functor from the Eilenberg-Moore resolution of P to the resolution $\mathsf{Free}_{\mathsf{Fn}} \uparrow \dashv \downarrow \mathsf{U}_{\mathsf{Fn}}$ of functorial algebras.*

Proof. By Definition 4.1, we need to show that $K^{\mathsf{EM}}_{\mathsf{Fn}}$ commutes with left and right adjoints of both resolutions: for right adjoints, we have $\mathsf{U}_{\Sigma+\Gamma\circ P}\langle X, \alpha \rangle = X = \downarrow \mathsf{U}_{\mathsf{Fn}} K^{\mathsf{EM}}_{\mathsf{Fn}} \langle X, \alpha \rangle$; and for left adjoints,

$$\begin{aligned} K^{\mathsf{EM}}_{\mathsf{Fn}}(\mathsf{Free}_{\Sigma+\Gamma\circ P} X) &= K^{\mathsf{EM}}_{\mathsf{Fn}} \langle (\Sigma + \Gamma \circ P)^* X,\ \mathsf{op}_X^{(\Sigma+\Gamma\circ P)^*} \rangle \\ &= \langle P, (\Sigma + \Gamma \circ P)^* X, \mathsf{in}, \mathsf{op}_X^{\cdots} \rangle \qquad \{P \cong G^* 0 \text{ and } (\Sigma + \Gamma \circ P) = I_P \text{ (Section 3.3)}\} \\ &\cong \langle G^* 0, (I_{G^*0})^* X, \mathsf{op}_0^{G^*}, \mathsf{op}_X^{(I_{G^*0})^*} \rangle \qquad \{\text{By Lemma 3.5}\} \\ &\cong \mathsf{Free}_{\mathsf{Fn}}(\uparrow X) \end{aligned}$$

and similarly for the actions on morphisms. Here we only have $K^{\mathsf{EM}}_{\mathsf{Fn}}\mathsf{Free}_{\Sigma+\Gamma\circ P}$ being isomorphic to $\mathsf{Free}_{\mathsf{Fn}}\Uparrow$ instead of a strict equality, since these two resolutions induce the monad P only up to isomorphism. To remedy this, one can slightly generalise the definition of comparison functors to take an isomorphism into account, but we leave it out here. □

4.6 Translating Functorial Algebras to Indexed Algebras

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

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

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

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

and $p_i = \alpha^G_{H^iX}\cdot\iota_1 : H^iX \to HH^iX$. On morphisms, $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ maps a morphism $\langle\tau : H \to H', f : X \to X'\rangle$ in $\mathtt{Fn\text{-}Alg}$ to $\sigma : H^iX \to H'^iX'$ 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 4.6. *Functor $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ is a comparison functor from the resolution $\mathsf{Free}_{\mathsf{Fn}}\Uparrow \dashv \Downarrow\mathsf{U}_{\mathsf{Fn}}$ of functorial algebras to the resolution $\mathsf{Free}_{\mathsf{Ix}}\upharpoonleft \dashv \downharpoonright\mathsf{U}_{\mathsf{Ix}}$ of indexed algebras.*

Proof. We need to show the required commutativities for $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ to be a comparison functor:

$$\Downarrow\mathsf{U}_{\mathsf{Fn}} \cong \downharpoonright\mathsf{U}_{\mathsf{Ix}}K^{\mathsf{Fn}}_{\mathsf{Ix}} \qquad \text{and} \qquad K^{\mathsf{Fn}}_{\mathsf{Ix}}\mathsf{Free}_{\mathsf{Fn}}\Uparrow \cong \mathsf{Free}_{\mathsf{Ix}}\upharpoonleft$$

First it is easy to see that it commutes with the right adjoints:

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

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

$$k^{\Sigma}_n = \left(\Sigma(P^+_X)_n = \Sigma PP^nX \xrightarrow{\mathsf{in}\cdot\iota_2} PP^nX\right)$$
$$k^{\Gamma\triangleleft}_n = \left((\Gamma\triangleleft P^+_X)_n = \Gamma PPP^nX \xrightarrow{\mathsf{in}\cdot\iota_3} PP^nX\right) \qquad k^{\triangleright}_n = \left((P^+_X)_n = PP^nX \xrightarrow{\mathsf{in}\cdot\iota_1} PPP^nX\right)$$

where $\mathsf{in} : Id + \Sigma\circ P + \Gamma\circ P\circ P \to P$ is the isomorphism between P and GP. Also by Lemma 3.5, we know that $\mathsf{Free}_{\mathsf{Fn}}\Uparrow X$ is isomorphic to the functorial algebra $\langle P, PX, \mathsf{in}, [\mathsf{in}\cdot\iota_2, \mathsf{in}\cdot\iota_3]\rangle$. Clearly $K^{\mathsf{Fn}}_{\mathsf{Ix}}\mathsf{Free}_{\mathsf{Fn}}\Uparrow$ and $\mathsf{FreeIx}\upharpoonleft$ agree on the carrier P^+_X. It can be checked that they agree on the structure maps and the action on morphisms too. □

Since comparison functors preserve interpretation (Lemma 4.3), 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.

Although the three kinds of algebras have the same expressivity in theory, they structure the interpretation of scoped operations in different ways: EM algebras impose no constraint on how the part of syntax enclosed by scopes is handled; indexed algebras demand them to be handled layer by layer but impose no coherent conditions between the layers; functorial algebras additionally force all inner layers must be handled in a uniform way by an endofunctor. On the whole, it is a trade-off *simplicity* and *structuredness*: EM algebras are the simplest for implementation, whereas the structuredness of functorial algebras make them easier to reason about. In the next section, we study an important reasoning principle, the *fusion law*, for the three kinds of algebras.

5 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 present the fusion law for functorial algebras, and as a case study and another technical contribution, we use the fusion law to show that the hybrid recursion scheme in [Piróg et al. 2018] is equivalent to interpreting with indexed algebras (Section 5.2).

5.1 Fusion Laws of Interpretation

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

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

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

Lemma 5.1 (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 every $D : \mathbb{D}$, $g : A \to RD$ and $f : RD \to X$, if there is some D' and $h : D \to D'$ in $\mathbb{D}$ such that $RD' = X$ and $Rh = f$, then*

$$f \cdot \mathit{handle}_D\ g = \mathit{handle}_{D'}\ (f \cdot g) \tag{30}$$

Proof. We have $f \cdot \mathit{handle}_D\ g = Rh \cdot R(\epsilon_D \cdot Lg) = R(h \cdot \epsilon_D \cdot Lg)$. Then by the naturality of the counit ϵ, $h \cdot \epsilon_D = \epsilon_{D'} \cdot LRh$. Thus $R(h \cdot \epsilon_D \cdot Lg) = R(\epsilon_{D'} \cdot L(Rh \cdot g)) = \mathit{handle}_{D'}\ (f \cdot g)$. □

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

Lemma 5.2 (Fusion for Functorial Algebras). *Let $\hat{\alpha}_1 = \langle H, X_1, \alpha_1^G, \alpha_2^I \rangle$ be a functorial algebra (Definition 3.1) and $g : A \to X_1$, $f : X_1 \to X_2$ be any morphisms in $\mathbb{C}$. If there is a functorial algebra $\hat{\alpha}_2 = \langle H_2, X_2, \alpha_2^G, \alpha_2^I \rangle$ and a functorial algebra morphism $\langle \sigma : H_1 \to H_2, h : X_1 \to X_2 \rangle$, then*

$$f \cdot \mathit{handle}_{\hat{\alpha}_1}\ g = \mathit{handle}_{\hat{\alpha}_2}\ (f \cdot g)$$

Remark 5.1. Although theoretically the three kinds of algebras for interpreting scoped operations have the same expressivity, it is usually easier to apply the fusion law to functorial algebras—the higher structuredness of functorial algebras makes it easier to show f is a homomorphism of functorial algebras compared to the other two kinds. This is another instance of the preference for structured programming over unstructured language features, in the same way as structured loops being favoured over `goto`, although they have the same expressivity in theory [Dijkstra 1968].

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Nat} + 1$$

$$\begin{aligned}
&\textbf{data}\ \mathit{IxAlg}\ \Sigma\ \Gamma\ a = \\
&\quad \mathit{IxAlg}\ \{\mathit{action} :: \forall n.\ \Sigma\ (a\ n) \to a\ n, \mathit{demote} :: \forall n.\ \Gamma\ (a\ (n+1)) \to a\ n, \mathit{promote} :: \forall n.\ a\ n \to a\ (n+1)\} \\
&\mathit{hfold} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{IxAlg}\ f\ g\ a \to \forall n :: \mathit{Nat}.\ \mathit{Prog}\ f\ g\ (a\ n) \to a\ n \\
&\mathit{hfold}\ \mathit{ixAlg}\ (\mathit{Return}\ x) \quad = x \\
&\mathit{hfold}\ \mathit{ixAlg}\ (\mathit{Call}\ \mathit{op}) \quad = \mathit{action}\ \mathit{ixAlg}\ (\mathit{fmap}\ (\mathit{hfold}\ \mathit{ixAlg})\ \mathit{op}) \\
&\mathit{hfold}\ \mathit{ixAlg}\ (\mathit{Enter}\ \mathit{scope}) = \mathit{demote}\ \mathit{ixAlg}\ (\mathit{fmap}\ (\mathit{hfold}\ \mathit{ixAlg} \cdot \mathit{fmap}\ (\mathit{promote}\ \mathit{ixAlg} \cdot \mathit{hfold}\ \mathit{ixAlg}))\ \mathit{scope})
\end{aligned}$$

Fig. 4. The hybrid fold for interpreting monad P with indexed algebras [Piróg et al. 2018]

In the rest of this section, we show an extended case study of using the fusion laws and comparison functors to reason about algebras of scoped operations. In particular, we prove that the *hfold* function (Figure 4) that Piróg et al. [2018] use to interpret scoped operations with indexed algebras indeed coincides coincides with interpretation *handle* with indexed algebras.

5.2 Case Study: Hybrid Fold

Although Piróg et al. [2018] propose the adjunction $(\mathtt{Free}_{\mathtt{Ix}}\ \upharpoonleft) \dashv (\downharpoonleft \mathtt{U}_{\mathtt{Ix}})$ for interpreting scoped operations with indexed algebras, they use the recursive function *hfold* (Figure 4) in their implementation to interpret P (13) with indexed algebras. Compared to a faithful implementation of *handle*, their *hfold* is more efficient since it skips transforming P to the free indexed algebra $(\downharpoonleft \mathtt{U}_{\mathtt{Ix}}\mathtt{Free}_{\mathtt{Ix}}\ \upharpoonleft)$ but directly work on P. Thus we call it a *hybrid fold* since it works on a syntactic structure P that is not freely generated by its type of algebras giving semantics.

While the definition of *hfold* is computationally intuitive, Piróg et al. [2018] did not provide formal justification for this recursive definition. In this subsection, we fill the gap by showing that *hfold* coincides with *handle* with indexed algebras. We divide the proof into three parts for clarity: after making the problem precise (Section 5.2.1), we first show that the *hfold* for an indexed algebra A is equivalently a catamorphism from P in $\mathsf{Endo}_f(\mathbb{C})$ (Section 5.2.2), which is then a special case of interpreting with functorial algebras (Section 5.2.3), and finally we translate this functorial algebra into the category $\mathtt{Ix\text{-}Alg}$ of indexed algebras using $K^{\mathtt{Fn}}_{\mathtt{Ix}}$, and show that it induces the same interpretation as the one from the indexed algebra A that we start with (Section 5.2.4).

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

$$\langle A : \mathbb{C}^{|\mathbb{N}|},\ a : \Sigma \circ A \to A,\ d : \Gamma \circ (\triangleleft A) \to A, p : A \to (\triangleleft A)\rangle \tag{31}$$

For notational convenience, we define a functor $S : |\mathbb{N}| \to |\mathbb{N}|$ such that $Sn = n + 1$, then $\triangleleft A = A \circ S$ since $(\triangleleft A)_n = A_{n+1} = A(Sn)$. With functor S, we can view p and d as $p : A \to A \circ S$ and $d : \Gamma \circ A \circ S \to A$. Then the recursive definition of *hfold* in Figure 4 can be understood as a morphism $h : P \circ A \to A$ in $\mathbb{C}^{|\mathbb{N}|}$ satisfying the equation

$$h = [h_1, h_2, h_3] \cdot (\mathsf{in}^\circ \circ A) \tag{32}$$

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

$$h_1 = \left(\mathit{Id} \circ A \xrightarrow{\mathsf{id}} A\right) \qquad h_2 = \left(\Sigma \circ P \circ A \xrightarrow{\Sigma \circ h} \Sigma \circ A \xrightarrow{a} A\right)$$

$$h_3 = \left(\Gamma \circ P \circ P \circ A \xrightarrow{\Gamma \circ P \circ h} \Gamma \circ P \circ A \xrightarrow{\Gamma \circ P \circ p} \Gamma \circ P \circ A \circ S \xrightarrow{\Gamma \circ h \circ S} \Gamma \circ A \circ S \xrightarrow{d} A\right)$$

In general, an equation in the form of (32) does *not* necessarily have a (unique) solution. Thus the semantics of the function *hfold* is not clear. We settle this problem with the following result.

Theorem 5.3 (Hybrid Folds Coincide with Interpretation). *There exists a unique solution to (32) and it coincides with the interpretation with indexed algebra A at level 0 (28):*

$$h_0 = handle_{\langle A,a,d,p\rangle}\ id : PA_0 \to A_0$$

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

5.2.2 Hybrid Fold Is an Adjoint Fold. The first step of our proof is to show the unique existence of the solution to (32) based on the observation that it is an *adjoint fold equation* [Hinze 2013] with the adjunction between *right Kan extension* [Mac Lane 1998] and composition with A. Hinze [2013] shows the following theorem stating that adjoint fold equations have a unique solution.

Theorem 5.4 (Mendler-style Adjoint Folds [Hinze 2013]). *Given any adjunction $L \dashv R : \mathbb{D} \to \mathbb{C}$, 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*

$$x = \Phi_{\mu G}(x) \cdot Lin^{\circ} \tag{33}$$

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 $h : PA \to A$ and $P = \mu G : \mathsf{Endo}_f(\mathbb{C})$, to apply this theorem to (32), we only need to (i) make $(-\circ A) : \mathsf{Endo}_f(\mathbb{C}) \to \mathbb{C}^{|\mathbb{N}|}$ a left adjoint and (ii) make $[h_1, h_2, h_3]$ in (32) an instance of $\Phi_P(h)$ for some natural transformation Φ:

- For (i), the functor $-\circ A$ is left adjoint to the *right Kan extension* along A, that is a functor $\mathsf{Ran}_A : \mathbb{C}^{|\mathbb{N}|} \to \mathsf{Endo}_f(\mathbb{C})$, which always exists when $\mathbb{C}$ is lfp.
- For (ii), we define a natural transformation $\Phi : \mathbb{C}^{|\mathbb{N}|}(-\circ A, A) \to \mathbb{C}^{|\mathbb{N}|}((G-)\circ A, A)$ such that for all $H : \mathsf{Endo}_f(\mathbb{C})$ and $f \in \mathbb{C}^{|\mathbb{N}|}(H\circ A, A)$, $\Phi_H(f) = [f_1, f_2, f_3]\cdot(\mathsf{in}^{\circ}\circ A)$ where

$$f_1 = \left(Id\circ A \xrightarrow{\mathsf{id}} A\right) \qquad f_2 = \left(\Sigma\circ H\circ A \xrightarrow{\Sigma\circ f} \Sigma\circ A \xrightarrow{a} A\right) \tag{34}$$
$$f_3 = \left(\Gamma\circ H\circ H\circ A \xrightarrow{\Gamma\circ H\circ f} \Gamma\circ H\circ A \xrightarrow{\Gamma\circ H\circ p} \Gamma\circ H\circ A\circ S \xrightarrow{\Gamma f} \Gamma\circ A\circ S \xrightarrow{d} A\right)$$

  It is immediate that *hfold* (32) is exactly $h = \Phi_P(h)\cdot(\mathsf{in}^{\circ}\circ A)$.

Then by Theorem 5.4 we have the following result.

Lemma 5.5 (Unique Existence of Hybrid Fold). *The recursive definition $h : PA \to A$ (32) of hybrid folds (Figure 4) has a unique solution:*

$$h = \left(PA \xrightarrow{(\!|\lfloor\Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor|\!)A} (\mathsf{Ran}_A A)A \xrightarrow{\epsilon_A} A\right) \tag{35}$$

where $\epsilon_A : (\mathsf{Ran}_A A)\circ A \to A$ is the counit, and $(\!|\lfloor\Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor|\!)$ is the catamorphism from the initial G-algebra P (13) to the G-algebra carried by $\mathsf{Ran}_A A$ with structure map $\lfloor\Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor$.

5.2.3 Catamorphism as Interpretation. We have shown that $hfold_A$ for all indexed algebras A uniquely exists, and now to show its coincidence with $handle_A$ (the second part of Theorem 5.3), we show that the hybrid fold coincides with *handle* with some *functorial* algebra, and then use the comparison functor $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ (Section 4.6) to translate this functorial algebra into an indexed algebra.

Consider the G-algebra in Lemma 5.5 carried by $\mathsf{Ran}_A A$ with structure map $\alpha^G = \lfloor\Phi_{\mathsf{Ran}_A A}(\epsilon_A)\rfloor$, we can trivially make it a functorial algebra $\hat{\alpha}$ by Proposition 3.3, hoping for applications of the fusion law of functorial algebras (Lemma 5.2):

$$\hat{\alpha} = J_{A_0}\langle \mathsf{Ran}_A A, \alpha^G\rangle = \langle \mathsf{Ran}_A A, (\mathsf{Ran}_A A)A_0, \alpha^G, \alpha^G\cdot[\iota_2, \iota_3]\rangle \tag{36}$$

The catamorphism to α^G and $handle_{\hat{\alpha}}$ are related by $(\!|\alpha^G|\!)_{A_0} = handle_{\hat{\alpha}}\ (\alpha^G_{A_0} \cdot \iota_1)$, which can be shown by checking that the formula (22) for computing $handle_{\hat{\alpha}}$ is exactly the defining equation of the catamorphism $(\!|\alpha^G|\!)_{A_0}$. Plugging the identity into (35), we obtain

$$h_0 = (\epsilon_A)_0 \cdot handle_{\hat{\alpha}}\ (\alpha^G_{A_0} \cdot \iota_1) \tag{37}$$

Now we have made some progress because the right-hand side takes exactly the form for applications of the fusion law—an interpretation followed by a morphism $(\epsilon_A)_0 : (\mathsf{Ran}_A A)A_0 \to A_0$.

In order to apply the fusion law, we need to make $(\epsilon_A)_0 : (\mathsf{Ran}_A A)A_0 \to A_0$ a functorial algebra homomorphism from $\hat{\alpha}$ (36). With some exploration, we can find a functorial algebra $\hat{\alpha}' = \langle \mathsf{Ran}_A A,\ A_0,\ \alpha^G,\ \alpha^I \rangle$ differing from $\hat{\alpha}$ by the second and fourth components, where

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

and a, d and p are the structure maps of indexed algebra A (31). It can be checked that $\langle \mathsf{id}, (\epsilon_A)_0 \rangle$ is a functorial algebra homomorphism from $\hat{\alpha}$ to $\hat{\alpha}'$. Thus by Lemma 5.2, we obtain

$$h_0 = handle_{\hat{\alpha}}\ \big((\epsilon_A)_0 \cdot (\alpha^G)_{A_0} \cdot \iota_1\big) = handle_{\hat{\alpha}'}\ \mathsf{id} : PA_0 \to A_0 \tag{38}$$

which means that the hybrid fold coincides with the interpretation with functorial algebra $\hat{\alpha}'$.

5.2.4 *Translating Back to Indexed Algebras.* The last step of our proof is translating the functorial algebra $\hat{\alpha}'$ back to an indexed algebra using comparison functor $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ (Section 4.6), 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 $K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'$ is an indexed algebra carried by $i \mapsto (\mathsf{Ran}_A A)^i A_0$ and by Lemma 4.3 and (38), $K^{\mathsf{Fn}}_{\mathsf{Ix}}$ preserves the induced interpretation:

$$h_0 = handle_{\hat{\alpha}'}\ \mathsf{id} = handle_{K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'}\ \mathsf{id} : PA_0 \to A_0 \tag{39}$$

What remains is to prove that $handle_{K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'}\ \mathsf{id} = handle_A\ \mathsf{id}$, and we show this by the fusion law (of indexed algebras): define a natural transformation $\tau : (\mathsf{Ran}_A A)^i A_0 \to A_i$ between $\mathbb{C}^{|\mathbb{N}|}$ functors by $\tau_0 = \mathsf{id} : (\mathsf{Ran}_A A)^0 A_0 \to A_0$ and

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

and it can be checked that τ is an indexed algebra homomorphism from $K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'$ to $\langle A, a, d, p \rangle$. Note that we have $\downharpoonleft \mathsf{U}_{\mathsf{Ix}}\tau$ equals $\mathsf{id} : A_0 \to A_0$, and by Lemma 4.3, we have that

$$h_0 = handle_{K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'}\ \mathsf{id} = \mathsf{id} \cdot handle_{K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'}\ \mathsf{id} = (\downharpoonleft \mathsf{U}_{\mathsf{Ix}})\tau \cdot handle_{K^{\mathsf{Fn}}_{\mathsf{Ix}}\hat{\alpha}'}\ \mathsf{id} = handle_A\ \mathsf{id}$$

This completes our proof of Theorem 5.3 saying that Piróg et al. [2018]'s *hfold* indeed correctly implements *handle* with indexed algebras.

To summarise, we first connected *hfold* to a functorial algebra using a Mendler-style adjoint fold, and then used the fusion law of functorial algebras to simplify it. Then it is translated to the category of indexed algebras, and we used the fusion law one more time there.

6 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] popularised 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 in an untyped language but not in a language with a simple type system. 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 4.1, an alternative view on Eilenberg-Moore algebras of scoped operations is regarding them as handlers of *algebraic* operations of signature $(\Sigma + \Gamma P)$. However, the functor $(\Sigma + \Gamma P)$ mentions the type P 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 P of computations, which is usually needed to concisely implement the component $\Gamma P X \to X$ 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 P, 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 $\mathtt{Free}_{\mathtt{Ix}} \dashv \mathtt{U}_{\mathtt{Ix}}$ of P via indexed algebras restores some modularity in the sense that

$$P \;\cong\; \rfloor\,(\Sigma - {+}\Gamma(\triangleleft -) + (\triangleright -))^{*}\,\lceil \;\cong\; \rfloor\,(\Sigma^{*} +_{\mathsf{Mnd}} (\Gamma(\triangleleft -) + (\triangleright -))^{*})\,\lceil$$

where $+_{\mathsf{Mnd}}$ 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. Possible directions towards this are the categorical framework of *equational systems* [Fiore and Hur 2009b] and syntactic frameworks like [Fiore and Mahmoud 2010] and [Staton 2013].

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 P 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 $handle_D\ g : GFX \longrightarrow A$ for some adjunction $F \dashv G$ can be understood as another way of generalising catamorphisms $(\mu D\ \cong)\ \mathsf{U}_D\mathsf{Free}_D 0 \longrightarrow A$ 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].

7 CONCLUSION

The motivation of this work is to develop a modular approach to the syntax and semantics of scoped operations. We believe our proposal, functorial algebras, is at a sweet spot in the trade-off between structuredness and simplicity, allowing practical examples of scoped operations to be programmed and reasoned about naturally, and implementable in modern functional languages such as Haskell and OCaml. We put our model of interpreting scoped operations and two existing models 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

J. Adamek and J. Rosicky. 1994. *Locally Presentable and Accessible Categories*. Cambridge University Press. https://doi.org/10.1017/CBO9780511600579

Jiří Adámek. 1974. Free algebras and automata realizations in the language of categories. *Commentationes Mathematicae Universitatis Carolinae* 015, 4 (1974), 589–602. http://eudml.org/doc/16649

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

Roland Backhouse, Marcel Bijsterveld, Rik van Geldrop, and Jaap van der Woude. 1995. Categorical fixed point calculus. In *Category Theory and Computer Science*, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179. https://doi.org/10.1007/3-540-60164-3_25

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

Michael Barr and Charles Wells. 1990. *Category theory for computing science*. Vol. 1. Prentice Hall New York.

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

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

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

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

Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM* 11, 3 (March 1968), 147–148. https://doi.org/10.1145/362929.362947

Marcelo Fiore and Chung-Kil Hur. 2009a. On the construction of free algebras for equational systems. *Theoretical Computer Science* 410, 18 (2009), 1704–1729. https://doi.org/10.1016/j.tcs.2008.12.052 Automata, Languages and Programming (ICALP 2007).

Marcelo Fiore and Chung-Kil Hur. 2009b. On the construction of free algebras for equational systems. *Theoretical Computer Science* 410, 18 (2009), 1704–1729. https://doi.org/10.1016/j.tcs.2008.12.052 Automata, Languages and Programming (ICALP 2007).

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, Christoph Lüth, Federico De Marchi, and John Power. 2003. Dualising Initial Algebras. *Mathematical. Structures in Comp. Sci.* 13, 2 (April 2003), 349–370. https://doi.org/10.1017/S0960129502003912

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

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. 1998. Prological Features in a Functional Setting — Axioms and Implementations. In *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming (FLOPS '98)* (Kyoto, Japan), Masahiko Sato and Yoshihito Toyama (Eds.). World Scientific, Singapore, New Jersey, London, Hong Kong, 98–122.

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

Martin Hyland and John Power. 2007. The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads. *Electronic Notes in Theoretical Computer Science* 172 (2007), 437–458. https://doi.org/10.1016/j.entcs.2007.02.019 Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

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

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic Foundations for Effect-Dependent Optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2103656.2103698

G.M. Kelly and A.J. Power. 1993. Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. *Journal of Pure and Applied Algebra* 89, 1 (1993), 163–179. https://doi.org/10.1016/0022-4049(93)90092-8

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

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.*, 94–105. https://doi.org/10.1145/2804302.2804319

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

Paul Blain Levy. 2003a. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science* 69 (2003), 248–271. https://doi.org/10.1016/S1571-0661(04)80568-1 CTCS'02, Category Theory and Computer Science.

Paul Blain Levy. 2003b. *Call-by-push-value: A Functional/Imperative Synthesis*. Vol. 2. Springer Netherlands. https://doi.org/10.1007/978-94-007-0954-6

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.

Ralph Matthes and Tarmo Uustalu. 2004. Substitution in non-wellfounded syntax with variable binding. *Theoretical Computer Science* 327, 1 (2004), 155–174. https://doi.org/10.1016/j.tcs.2004.07.025 Selected Papers of CMCS '03.

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.

A John Power. 1999. Enriched lawvere theories. *Theory and Applications of Categories* 6, 7 (1999), 83–93.

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.

Edmund Robinson. 2002. Variations on Algebra: Monadicity and Generalisations of Equational Theories. 13, 3 (2002), 308–326. https://doi.org/10.1007/s001650200014

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

Mike Spivey. 1990. A functional theory of exceptions. *Science of Computer Programming* 14, 1 (1990), 25–42. https://doi.org/10.1016/0167-6423(90)90056-J

Ian Stark. 2008. Free-algebra models for the π-calculus. *Theoretical Computer Science* 390, 2 (2008), 248–270. https://doi.org/10.1016/j.tcs.2007.09.024 Foundations of Software Science and Computational Structures.

Sam Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. 519–519. https://doi.org/10.1109/LICS.2013.58

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

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 24–52. https://doi.org/10.5555/647698.734146

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).