Hefty Algebras

Modular Elaboration of Higher-Order Algebraic Effects

CASPER BACH POULSEN, Delft University of Technology, Netherlands
CAS VAN DER REST, Delft University of Technology, Netherlands

Algebraic effects and handlers is an increasingly popular approach to programming with effects. An attraction of the approach is its modularity: effectful programs are written against an interface of declared operations, which allows the implementation of these operations to be defined and refined without changing or recompiling programs written against the interface. However, higher-order operations (i.e., operations that take computations as arguments) break this modularity. While it is possible to encode higher-order operations by elaborating them into more primitive algebraic effects and handlers, such elaborations are typically not modular. In particular, operations defined by elaboration are typically not a part of any effect interface, so we cannot define and refine their implementation without changing or recompiling programs. To resolve this problem, a recent line of research focuses on developing new and improved effect handlers. In this paper we present a (surprisingly) simple alternative solution to the modularity problem with higher-order operations: we modularize the previously non-modular elaborations commonly used to encode higher-order operations. Our solution is as expressive as the state of the art in effects and handlers.

CCS Concepts: • **Software and its engineering → Semantics**; • **Theory of computation → Type structures**; **Program specifications**; *Denotational semantics.*

Additional Key Words and Phrases: Algebraic Effects, Modularity, Reuse, Agda, Dependent Types

ACM Reference Format:
Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL, Article 62 (January 2023), 31 pages. https://doi.org/10.1145/3571255

1 INTRODUCTION

Defining abstractions for programming with side effects is a research question with a long and rich history. The goal is to define an interface of (possibly) side effecting operations where the interface encapsulates and hides irrelevant operational details about the operations and their side effects. Such encapsulation makes it easy to refactor, optimize, or even change the behavior of a program, by changing the implementation of the interface.

Monads [Moggi 1989b] have long been the preferred solution to this research question. However, *algebraic effects and handlers* [Plotkin and Pretnar 2009] are emerging as an attractive alternative solution, due to the modularity benefits that they provide. However, these modularity benefits do not apply to many common operations that take computations as arguments.

Authors' addresses: Casper Bach Poulsen, Delft University of Technology, Netherlands, c.b.poulsen@tudelft.nl; Cas van der Rest, Delft University of Technology, Netherlands, c.r.vanderrest@tudelft.nl.

2475-1421/2023/1-ART62
https://doi.org/10.1145/3571255

1.1 Background: Algebraic Effects and Handlers

To understand the benefits of algebraic effects and handlers and the modularity problem with operations that take computations as parameters, we give a brief introduction to algebraic effects, based on the effect handlers tutorial by Pretnar [2015]. Readers familiar with algebraic effects and handlers are encouraged to skim the code examples in this subsection and read its final paragraph.

Consider a simple operation *out* for output which takes a string as argument and returns the unit value. Using algebraic effects and handlers its type is:

$$out : String \to () \,!\, Output$$

Here *Output* is the *effect* of the operation. In general $A \,!\, \Delta$ is a computation type where A is the return type and Δ is a *row* (i.e., unordered sequence) of *effects*, where an *effect* is a label associated with a set of operations. A computation of type $A \,!\, \Delta$ may *only* use operations associated with an effect in Δ. An effect can generally be associated with multiple operations (but not the other way around); however, the simple *Output* effect that we consider is only associated with the operation *out*. Thus $() \,!\, Output$ is the type of a computation which may call the *out* operation.

We can think of *Output* as an interface that specifies the parameter and return type of *out*. The implementation of such an interface is given by an *effect handler*. An effect handler defines how to interpret operations in the execution context they occur in. The type of an effect handler is $A \,!\, \Delta \implies B \,!\, \Delta'$, where Δ is the row of effects before applying the handler and Δ' is the row after. For example, here is the type of an effect handler for *Output*:

$$hOut : A \,!\, Output, \Delta \Rightarrow (A \times String) \,!\, \Delta$$

The *Output* effect is being handled, so it is only present in the effect row on the left.[1] As the type suggests, this handler handles *out* operations by accumulating a string of output. Below is the handler of this type:

$$\begin{aligned} hOut = \textbf{handler}\ \{\ & (\textbf{return}\ x) \mapsto \textbf{return}\ (x, \text{“”}) \\ & (out\ s; k) \mapsto \textbf{do}\ (y, s') \leftarrow k\ (); \textbf{return}\ (y, s +\!\!+ s')\ \} \end{aligned}$$

The **return** case of the handler says that, if the computation being handled terminates normally with a value x, then we return a pair of x and the empty string. The case for *out* binds a variable s for the string argument of the operation, but also a variable k representing the *execution context* (or *continuation*). Invoking an operation suspends the program and its execution context up-to the nearest handler of the operation. The handler can choose to re-invoke the suspended execution context (possibly multiple times). The handler case for *out* above always invokes k once. Since k represents an execution context that includes the current handler, calling k gives a pair of a value y and a string s', representing the final value and output of the execution context. The result of handling *out* s is then y and the current output (s) plus the output of the rest of the program (s').

In general, a computation $m : A \,!\, \Delta$ can only be run in a context that provides handlers for each effect in Δ. To this end, if $\Delta = \Delta_1, \Delta_2$ and $h : A \,!\, \Delta_1 \implies B \,!\, \Delta_1'$, then the expression $(\textbf{with}\ h\ \textbf{handle}\ m) : B \,!\, \Delta_1', \Delta_2$ runs m in the context of the handler h. For example, consider:

$$\begin{aligned} & hello : () \,!\, Output \\ & hello = out\ \text{“Hello”};\ out\ \text{“ world!”} \end{aligned}$$

Using this, we can run *hello* in a scope with the handler *hOut* to compute the following result:

$$(\textbf{with}\ hOut\ \textbf{handle}\ hello) \equiv ((), \text{“Hello world!”})$$

[1] *Output* could occur in Δ too. This raises the question: which *Output* effect does a given handler actually handle? We refer to the literature for answers to this question; see, e.g., the row treatment of Morris and McKinna [2019], the *effect lifting* of Biernacki et al. [2018], and the *effect tunneling* of Zhang and Myers [2019].

An attractive feature of algebraic effects and handlers is that programs such as *hello* are defined *independently* of how the effectful operations they use are implemented. This makes it is possible to refine, refactor, or even change the meaning of operations without having to modify the programs that use them. For example, we can refine the meaning of *out without modifying the hello program*, by using a different handler *hOut′* which prints output to the console. However, some operations are challenging to express in a way that provides these modularity benefits.

1.2 The Modularity Problem with Higher-Order Operations

Algebraic effects and handlers provide limited support for operations that accept computations as arguments (sometimes called *higher-order operations*). The limitation is subtle but follows from how handler cases are typed. Following Plotkin and Pretnar [2009]; Pretnar [2015], the left and right hand sides of handler cases are typed as follows:

$$\textbf{handler}\,\{\cdots\,(op\;\underbrace{v}_{A}\;;\;\underbrace{k}_{B\to C\,!\,\Delta'}\;)\mapsto\underbrace{c}_{C\,!\,\Delta'},\cdots\}$$

Here it is only k whose type is compatible with the right hand side. In theory, the parameter type v would also be compatible if $A = C\,!\,\Delta'$. However, encoding computations as parameters in this way is non-modular. The reason is that effect handlers are not applied recursively to parameters of operations [Plotkin and Pretnar 2009; Pretnar 2015]; i.e., if h handles operations other than op, then

$$\textbf{with}\;h\;\textbf{handle}\;(\textbf{do}\;x\leftarrow op\;v;m)\;\equiv\;\textbf{do}\;x\leftarrow op\;v;(\textbf{with}\;h\;\textbf{handle}\;m)$$

This implies that the only way to ensure that v has type $A = C\,!\,\Delta'$ whose effects match the context of the operation (e.g., $k : B \to C\,!\,\Delta'$), is to apply handlers of higher-order effect encodings (such as op) before applying other handlers (such as h). In turn, this means that programs can contain at most one higher-order effect encoded in this way (otherwise, which handler do we apply first?). Consequently, encoding computation parameters in terms of the value v carried by an operation does not support modular definition, composition, and handling of higher-order effects.

A consequence of this handler typing restriction is that algebraic effects and handlers only support higher-order operations whose computation parameters are *continuation-like*. In particular, for any operation $op : A\,!\,\Delta \to \cdots \to A\,!\,\Delta \to A\,!\,\Delta$ and any $m_1, \ldots, m_n$ and k,

$$\textbf{do}\;x\leftarrow(op\;m_1\ldots m_n);k\;x\;\equiv\;op\;(\textbf{do}\;x_1\leftarrow m_1;k\;x_1)\ldots(\textbf{do}\;x_n\leftarrow m_n;k\;x_n)\qquad(\dagger)$$

This property, known as the *algebraicity property* [Plotkin and Power 2003], says that the computation parameter values $m_1, \ldots, m_n$ are only ever run in a way that *directly* passes control to k. Such operations can without loss of generality or modularity be encoded as operations *without computation parameters*; e.g., $op\;m_1\ldots m_n = \textbf{do}\;x\leftarrow op'\;();select\;x$ where $op' : () \to D^n\,!\,\Delta$ and $select : D^n \to A\,!\,\Delta$ is a function that chooses between n different computations using a data type D^n whose constructors are $d_1, \ldots, d_n$ such that $select\;d_i = m_i$ for $i = 1..n$. Some higher-order operations obey the algebraicity property; many do not. Examples of operations that do not include:

- Exception handling: let $catch\;m_1\;m_2$ be an operation that handles exceptions thrown during evaluation of computation m_1 by running m_2 instead, and $throw$ be an operation that throws an exception. These operations are not algebraic. For example,
$$\textbf{do}\;(catch\;m_1\;m_2);throw\;\not\equiv\;catch\;(\textbf{do}\;m_1;throw)\;(\textbf{do}\;m_2;throw)$$
- Local binding (the *reader monad* [Jones 1995]): let ask be an operation that reads a local binding, and $local\;r\;m$ be an operation that makes r the current binding in computation m. Observe:
$$\textbf{do}\;(local\;r\;m);ask\;\not\equiv\;local\;r\;(\textbf{do}\;m;ask)$$

- Logging with filtering (an extension of the *writer monad* [Jones 1995]): let *out s* be an operation for logging a string, and *censor f m* be an operation for post-processing the output of computation *m* by applying $f : \mathit{String} \to \mathit{String}$.[2] Observe:

$$\mathbf{do}\ (\mathit{censor}\ f\ m); \mathit{out}\ s \not\equiv \mathit{censor}\ f\ (\mathbf{do}\ m; \mathit{out}\ s)$$

It is, however, possible to elaborate higher-order operations into more primitive effects and handlers. For example, *censor* can be elaborated into an inline handler application of *hOut*:

$$\begin{aligned}&\mathit{censor} : (\mathit{String} \to \mathit{String}) \to A\,!\,\mathit{Output}, \Delta \to A\,!\,\mathit{Output}, \Delta\\ &\mathit{censor}\ f\ m = \mathbf{do}\ (x, s) \leftarrow (\mathbf{with}\ \mathit{hOut}\ \mathbf{handle}\ m);\ \mathit{out}\ (f\ s);\ \mathbf{return}\ x\end{aligned}$$

The other higher-order operations above can be defined in a similar manner.

Elaborating higher-order operations into standard algebraic effects and handlers as illustrated above is a key use case that effect handlers were designed for [Plotkin and Pretnar 2009]. However, elaborating operations in this way means the operations are not a part of any effect interface. So, unlike plain algebraic operations, the only way to refactor, optimize, or change the semantics of higher-order operations defined in this way is to modify or copy code. In other words, we forfeit one of the key attractive modularity features of algebraic effects and handlers.

This modularity problem with higher-order effects (i.e., effects with higher-order operations) was first observed by Wu et al. [2014] who proposed *scoped effects and handlers* [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] as a solution. Scoped effects and handlers have similar modularity benefits as algebraic effects and handlers, but works for a wider class of effects, including many higher-order effects. However, van den Berg et al. [2021] recently observed that operations that defer computation, such as evaluation strategies for λ application or *(multi-)staging* [Taha and Sheard 2000], are beyond the expressiveness of scoped effects. Therefore, van den Berg et al. [2021] introduced another flavor of effects and handlers that they call *latent effects and handlers*.

In this paper we present a (surprisingly) simple alternative solution to the modularity problem with higher-order effects, which only uses standard effects and handlers and off-the-shelf generic programming techniques known from, e.g., *data types à la carte* [Swierstra 2008].

1.3 Solving the Modularity Problem: Elaboration Algebras

We propose to define elaborations such as *censor* from § 1.2 in a modular way. To this end, we introduce a new type of *computations with higher-order effects* which can be modularly elaborated into computations with only standard algebraic effects:

$$A \mathbin{!!} H \xrightarrow{\mathit{elaborate}} A\,!\,\Delta \xrightarrow{\mathit{handle}} \mathit{Result}$$

Here $A \mathbin{!!} H$ is a computation type where A is a return type and H is a row comprising both algebraic and higher-order effects. The idea is that the higher-order effects in the row H are modularly elaborated into the row Δ. To achieve this, we define *elaborate* such that it can be modularly composed from separately defined elaboration cases, which we call elaboration *algebras* (for reasons we explain in § 3). Using $A \mathbin{!!} H \Rrightarrow A\,!\,\Delta$ as the type of elaboration algebras that elaborate the higher-order effects in H to Δ, we can modularly compose any pair of elaboration algebras $e_1 : A \mathbin{!!} H_1 \Rrightarrow A\,!\,\Delta$ and $e_2 : A \mathbin{!!} H_2 \Rrightarrow A\,!\,\Delta$ into an algebra $e_{12} : A \mathbin{!!} H_1, H_2 \Rrightarrow A\,!\,\Delta$.[3]

Elaboration algebras are as simple to define as non-modular elaborations such as *censor* (§ 1.2). For example, here is the elaboration algebra for the higher-order *Censor* effect whose only associated

[2] The *censor* operation is a variant of the function by the same name the widely used Haskell `mtl` library: https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Writer-Lazy.html

[3] Readers familiar with data types à la carte [Swierstra 2008] may recognize this as algebra composition.

operation is the higher-order operation $censor_{op} : (String \to String) \to A \mathbin{!!} H \to A \mathbin{!!} H$:

$$\begin{aligned} &eCensor : A \mathbin{!!} Censor \Rrightarrow A \mathbin{!} Output, \Delta \\ &eCensor\ (censor_{op}\ f\ m;\ k) = \mathbf{do}\ (x, s) \leftarrow (\mathbf{with}\ hOut\ \mathbf{handle}\ m);\ out\ (f\ s);\ k\ x \end{aligned}$$

The implementation of *eCensor* is essentially the same as *censor*. There are two main differences. First, elaboration happens in-context, so the value yielded by the elaboration is passed to the context (or continuation) k. Second, and most importantly, programs that use the $censor_{op}$ operation are now programmed against the interface given by *Censor*, meaning programs do not (and *cannot*) make assumptions about how $censor_{op}$ is elaborated. As a consequence, we can modularly refine the elaboration of higher-order operations such as $censor_{op}$, without modifying the programs that use the operations. For example, the following program censors and replaces "Hello" with "Goodbye":[4]

$$\begin{aligned} &censorHello : () \mathbin{!!} Censor, Output \\ &censorHello = censor_{op}\ (\lambda s.\ \mathbf{if}\ (s \equiv \text{"Hello"})\ \mathbf{then}\ \text{"Goodbye"}\ \mathbf{else}\ s)\ hello \end{aligned}$$

Say we have a handler $hOut' : (String \to String) \to A \mathbin{!} Output, \Delta \Rightarrow (A \times String) \mathbin{!} \Delta$ which handles each operation $out\ s$ by pre-applying a censor function $(String \to String)$ to s before emitting it. Using this handler, we can give an alternative elaboration of $censor_{op}$ which post-processes output strings *individually*:

$$\begin{aligned} &eCensor' : A \mathbin{!!} Censor \Rrightarrow A \mathbin{!} Output, \Delta \\ &eCensor'\ (censor_{op}\ f\ m;\ k) = \mathbf{do}\ x \leftarrow (\mathbf{with}\ hOut'\ f\ \mathbf{handle}\ m);\ out\ s;\ k\ x \end{aligned}$$

In contrast, *eCensor* applies the censoring function $(String \to String)$ to the batch output of the computation argument of a $censor_{op}$ operation. The batch output of *hello* is "Hello world!" which is unequal to "Hello", so *eCensor* leaves the string unchanged. On the other hand, *eCensor′* censors the individually output "Hello":

$$\begin{aligned} &\mathbf{with}\ hOut\ \mathbf{handle}\ (\mathbf{with}\ eCensor\ \mathbf{elaborate}\ censorHello) \equiv ((), \text{"Hello world!"}) \\ &\mathbf{with}\ hOut\ \mathbf{handle}\ (\mathbf{with}\ eCensor'\ \mathbf{elaborate}\ censorHello) \equiv ((), \text{"Goodbye world!"}) \end{aligned}$$

Higher-order operations now have the same modularity benefits as algebraic operations.

1.4 Contributions

This paper formalizes the ideas sketched in this introduction by shallowly embedding them in Agda. However, the ideas transcend Agda. Similar shallow embeddings can be implemented in other dependently typed languages, such as Idris [Brady 2013a]; but also in less dependently typed languages like Haskell, OCaml, or Scala.[5] By working a dependently typed language we can state algebraic laws about interfaces of effectful operations, and prove that implementations of the interfaces respect the laws. We make the following technical contributions:

- § 2 describes how to encode algebraic effects in Agda, revisits the modularity problem with higher-order operations, and summarizes how scoped effects and handlers address the modularity problem, for some (*scoped* operations) but not all higher-order operations.
- § 3 presents our solution to the modularity problem with higher-order operations. Our solution is to (1) type programs as *higher-order effect trees* (which we dub *hefty trees*), and (2) build modular elaboration algebras for folding hefty trees into algebraic effect trees and handlers. The computations of type $A \mathbin{!!} H$ discussed in § 1.3 correspond to hefty trees, and the elaborations of type $A \mathbin{!!} H \Rrightarrow A \mathbin{!} \Delta$ correspond to hefty algebras.

[4] This program relies on the fact that it is generally possible to lift computation $A \mathbin{!} \Delta$ to $A \mathbin{!!} H$ when $\Delta \subseteq H$.
[5] The artifact accompanying this paper [Bach Poulsen and Reinders 2023] contains a shallow embedding of elaboration algebras in Haskell.

- § 4 shows that hefty algebras support formal reasoning on a par with algebraic effects and handlers, by verifying algebraic laws of higher-order effects for exception catching.
- § 5 presents examples of how to define hefty algebras for common higher-order effects from the literature on effect handlers.

§ 6 discusses related work and § 7 concludes. An artifact containing the code of the paper and a Haskell embedding of the same ideas is available online [Bach Poulsen and Reinders 2023].

2 ALGEBRAIC EFFECTS AND HANDLERS IN AGDA

This section describes how to encode algebraic effects and handlers in Agda. We do not assume familiarity with Agda and explain Agda specific notation in footnotes. §§ 2.1 to 2.4 defines algebraic effects and handlers; § 2.5 revisits the problem of defining higher-order effects using algebraic effects and handlers; and § 2.6 discusses how scoped effects [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] solves the problem for some (*scoped* operations) but not all higher-order operations.

2.1 Algebraic Effects and The Free Monad

We encode algebraic effects in Agda by representing computations as an abstract syntax tree given by the *free monad* over an *effect signature*. Such effect signatures are traditionally [Awodey 2010; Kammar et al. 2013; Kiselyov and Ishii 2015; Swierstra 2008; Wu et al. 2014] given by a *functor*; i.e., a type of kind Set → Set together with a (lawful) mapping function.[6] In our Agda implementation, effect signature functors are defined by giving a *container* [Abbott et al. 2003, 2005]. Each container corresponds to a value of type Set → Set that is both *strictly positive*[7] and *universe consistent*[8] [Martin-Löf 1984], meaning they are a constructive approximation of endofunctors on Set. Using containers, effect signatures are given by a (dependent) record type:[9] [10]

```
record Effect : Set₁ where
  field Op  : Set
        Ret : Op → Set
```

Here, Op is a type of operations, and Ret defines the *return type* of each operation of type Op.

As discussed in the introduction, computations may use multiple different effects. We use the co-product of effect signature functors to encode rows of effects:[11] [12]

```
_⊕_ : Effect → Effect → Effect
Op  (Δ₁ ⊕ Δ₂) = Op Δ₁ ⊎ Op Δ₂
Ret (Δ₁ ⊕ Δ₂) = [ Ret Δ₁ , Ret Δ₂ ]
```

We compute the co-product of two effect signatures by taking the disjoint sum of their operations and combining the return type mappings pointwise. The effect $\Delta_1 \oplus \Delta_2$ corresponds to the row union denoted as Δ_1, Δ_2 in the introduction.

[6] Set is the type of types in Agda. More generally, functors mediate between different *categories*. For simplicity, this paper only considers endofunctors on Set.
[7] https://agda.readthedocs.io/en/v2.6.2.2/language/positivity-checking.html
[8] https://agda.readthedocs.io/en/v2.6.2.2/language/universe-levels.html
[9] https://agda.readthedocs.io/en/v2.6.2.2/language/record-types.html
[10] The type of effect rows has type Set_1 instead of Set. To prevent logical inconsistencies, Agda has a hierarchy of types where $\mathsf{Set} : \mathsf{Set}_1$, $\mathsf{Set}_1 : \mathsf{Set}_2$, etc.
[11] The _⊕_ function uses *copattern matching*: https://agda.readthedocs.io/en/v2.6.2.2/language/copatterns.html. The Op line defines how to compute the Op field of the record produced by the function; and similarly for the Ret line.
[12] _⊎_ is a *disjoint sum* type from the Agda standard library. It has two constructors, $\mathsf{inj}_1 : A \to A \uplus B$ and $\mathsf{inj}_2 : B \to A \uplus B$. The [_,_] function (also from the Agda standard library) is the *eliminator* for the disjoint sum type. Its type is $[\_,\_] : (A \to X) \to (B \to X) \to (A \uplus B) \to X$.

The syntax of computations with effects Δ is given by the free monad over Δ. Following Hancock and Setzer [2000] and Kiselyov and Ishii [2015], we encode the free monad as follows:

```
data Free (Δ : Effect) (A : Set) : Set where
  pure   : A                                  → Free Δ A
  impure : (op : Op Δ) (k : Ret Δ op → Free Δ A) → Free Δ A
```

Here, pure is a computation with no side-effects, whereas impure is an operation (*op* : Op Δ) whose continuation (*k* : Ret Δ *op* → Free Δ *A*) expects a value of the return type of the operation. To see how we can represent programs using this data type, it is instructional to look at an example.

Example 2.1. The data type on the left below defines an operation for outputting a string. On the right is its corresponding effect signature.

```
data OutOp : Set where
  out : String → OutOp
```

```
Output : Effect
Op  Output          = OutOp
Ret Output (out s) = ⊤
```

The effect signature on the right says that out returns a unit value (⊤ is the unit type). Using this, we can write a simple hello world corresponding to the *hello* program from § 1:

```
hello : Free Output ⊤
hello = impure (out "Hello") (λ _ → impure (out " world!") (λ x → pure x))
```

§ 2.1 shows how to make this program more readable by using monadic **do** notation.

The hello program above makes use of just a single effect. Say we want to use another effect, Throw, with a single operation, throw, which represents throwing an exception (therefore having the empty type ⊥ as its return type):

```
data ThrowOp : Set where
  throw : ThrowOp
```

```
Throw : Effect
Op  Throw = ThrowOp
Ret Throw throw = ⊥
```

Programs that use multiple effects, such as Output and Throw, are unnecessarily verbose. For example, consider the following program which prints two strings before throwing an exception:[13]

```
hello-throw : Free (Output ⊕ Throw) A
hello-throw = impure (inj₁ (out "Hello")) (λ _ →
                impure (inj₁ (out " world!")) (λ _ →
                  impure (inj₂ throw) ⊥-elim))
```

To reduce syntactic overhead, we use *row insertions* and *smart constructors* [Swierstra 2008].

2.2 Row Insertions and Smart Constructors

A row insertion Δ ∼ Δ₀ ▸ Δ′ is a data type representing a witness that Δ is the effect row resulting from inserting Δ₀ somewhere in Δ′:

```
data _~_▸_ : Effect → Effect → Effect → Set₁ where
  insert :                (Δ₀ ⊕ Δ′) ~ Δ₀ ▸ Δ′
  sift   : (Δ ~ Δ₀ ▸ Δ′) → ((Δ₁ ⊕ Δ) ~ Δ₀ ▸ (Δ₁ ⊕ Δ′))
```

The insert constructor represents a witness that Δ₀ is inserted in front of Δ′, whereas sift witnesses that Δ₀ is inserted into the row Δ₁ ⊕ Δ′ by inserting Δ₀ somewhere in Δ′.

[13] ⊥-elim is the eliminator for the empty type, encoding the *principle of explosion*: ⊥-elim : ⊥ → *A*.

Using row insertions we can coerce effects into larger ones, and define smart constructors like:

```
`out : ⦃ Δ ∼ Output ▸ Δ′ ⦄ → String → Free Δ ⊤
```

We refer to Bach Poulsen and Reinders [2023] for the full implementation of `out. The double brace wrapped row insertion parameter of `out tells us that the Output effect is a part of the row Δ. The smart constructor uses this witness to coerce an out operation into Δ. This allows `out to be used in any program that has at least the Output effect.

The double braces in ⦃ Δ ∼ Output ▸ Δ′ ⦄ declares the row insertion witness as an *instance argument* of `out. Instance arguments in Agda are conceptually similar to type class constraints in Haskell: when we call `out, Agda will attempt to automatically find a witness of the right type, and implicitly pass this as an argument.[14] By declaring the row insertion constructors insert and sift as instances, Agda is able to construct insertion witnesses for us automatically in most cases.[15]

2.3 Fold and Monadic Bind for Free

Since Free Δ is a monad, we can sequence computations using *monadic bind*, which is naturally defined in terms of the fold over Free.

```
fold : (A → B) → Alg Δ B → Free Δ A → B
fold g a (pure x)       = g x
fold g a (impure op k) = a op (fold g a ∘ k)
```

```
Alg : (Δ : Effect) (A : Set) → Set
Alg Δ A = (op : Op Δ) (k : Ret Δ op → A) → A
```

Besides the input computation to be folded (last parameter), the fold is parameterized by a function $A \to B$ (first parameter) which folds a pure computation, and an *algebra* Alg Δ A (second parameter) which folds an impure computation. We call the latter an algebra because it corresponds to an F-algebra [Arbib and Manes 1975; Pierce 1991] over the signature functor of Δ, denoted F_Δ. That is, a tuple (A, α) where A is an object called the *carrier* of the algebra, and α a morphism $F_\Delta(A) \to A$. Using fold, monadic bind for the free monad is defined as follows:

```
_>>=_ : Free Δ A → (A → Free Δ B) → Free Δ B
m >>= g = fold g impure m
```

Intuitively, $m \gg\!= g$ concatenates g to all the leaves in the computation m.

Example 2.2. By implementing a smart constructor `throw : ⦃ Δ ∼ Throw ▸ Δ′ ⦄ → Free Δ A for throw, our example program from before becomes more readable:

```
hello-throw₁ : ⦃ Δ ∼ Output ▸ Δ₁ ⦄ → ⦃ Δ ∼ Throw ▸ Δ₂ ⦄ → Free Δ A
hello-throw₁ = do `out "Hello"; `out "  world!"; `throw
```

This illustrates how we use the free monad to write effectful programs against an interface given by an effect signature. Next, we define *effect handlers*.

2.4 Effect Handlers

An effect handler implements the interface given by an effect signature, and defines how to interpret the syntactic operations associated with an effect. Like monadic bind, effect handlers can be defined as a fold over the free monad. The following type of *parameterized handlers* defines how to fold respectively pure and impure computations:

[14] For more details on how instance argument resolution works, see the Agda documentation: https://agda.readthedocs.io/en/v2.6.2.2/language/instance-arguments.html

[15] The two constructors for row insertion are *overlapping*, which will cause Agda instance resolution to fail unless we enable the option `--overlapping-instances`. The examples in this paper type check in Agda 2.6.2.2 using this option.

```
record ⟨_!_⇒_⇒_!_⟩ (A : Set) (Δ : Effect) (P : Set) (B : Set) (Δ′ : Effect) : Set₁ where
  field ret : A → P → Free Δ′ B
        hdl : Alg Δ (P → Free Δ′ B)
```

A handler of type $\langle\ A\ !\ \Delta \Rightarrow P \Rightarrow B\ !\ \Delta'\ \rangle$ is parameterized in the sense that it turns a computation of type Free Δ A into a parameterized computation of type $P \to$ Free Δ' B. The following function does so by folding using ret, hdl, and a function to-front : $\{\!|\ w : \Delta \sim \Delta_0 \blacktriangleright \Delta'\ |\!\} \to$ Free Δ A → Free $(\Delta_0 \oplus \Delta')$ A , whose implementation can be found in the artifact [Bach Poulsen and Reinders 2023].

```
given_handle_ : {|| Δ ~ Δ₀ ▸ Δ′ ||} → ⟨ A ! Δ₀ ⇒ P ⇒ B ! Δ′ ⟩ → Free Δ A → (P → Free Δ′ B)
given h handle m = fold
  (ret h)
  [ hdl h , (λ op k p → impure op (λ x → k x p)) ]
  (to-front m)
```

Comparing with the syntax we used to explain algebraic effects and handlers in the introduction, the ret field corresponds to the **return** case of the handlers from the introduction, and hdl corresponds to the cases that define how operations are handled. The parameterized handler type $\langle\ A\ !\ \Delta \Rightarrow P \Rightarrow B\ !\ \Delta'\ \rangle$ corresponds to the type $A\ !\ \Delta, \Delta' \Rightarrow P \to B\ !\ \Delta'$, and given h handle m corresponds to **with** h **handle** m.

Using this type of handler, the *hOut* handler from the introduction can be defined as follows:

```
hOut : ⟨ A ! Output ⇒ ⊤ ⇒ (A × String) ! Δ ⟩
ret hOut x _ = pure (x , "")
hdl hOut (out s) k p = do (x , s′) ← k tt p; pure (x , s ++ s′)
```

The handler *hOut* in § 1.1 did not bind any parameters. However, since we are encoding it as a *parameterized* handler, hOut now binds a unit typed parameter. Besides this difference, the handler is the same as in § 1.1. We can use the hOut handler to run computations. To this end, we introduce a Nil effect with no associated operations which we will use to indicate where an effect row ends:

```
Nil : Effect
Op Nil = ⊥
Ret Nil = ⊥-elim
```

```
un : Free Nil A → A
un (pure x) = x
```

Using these, we can run a simple hello world program:[16]

```
hello′ : Free (Output ⊕ Nil) ⊤
hello′ = do
  `out "Hello"; `out " world!"
```

```
test-hello : un ((given hOut handle hello′) tt)
           ≡ (tt , "Hello world!")
test-hello = refl
```

An example of an effect handler that makes use of parameterized (as opposed to unparameterized) handlers, is the state effect. Figure 1 declares and illustrates how to handle such an effect with operations for reading (get) and changing (put) the state of a memory cell holding a natural number.

2.5 The Modularity Problem with Higher-Order Effects, Revisited

§ 1.2 described the modularity problem with higher-order effects, using a higher-order operation that interacts with output as an example. In this section we revisit the problem, framing it in terms of the definitions introduced in the previous section, using a different effect whose interface is summarized

[16] The refl constructor is from the Agda standard library, and witnesses that a propositional equality (≡) holds.

```
data StateOp : Set where
  get :       StateOp
  put : ℕ → StateOp
```

```
State : Effect
Op State = StateOp
Ret State get     = ℕ
Ret State (put n) = ⊤
```

```
hSt : ⟨ A ! State ⇒ ℕ ⇒ (A × ℕ) ! Δ′ ⟩
ret hSt x s = pure (x , s)
hdl hSt (put m) k n = k tt m
hdl hSt get     k n = k n n
```

```
`incr : ⦃ Δ ∼ State ▸ Δ′ ⦄ → Free Δ ⊤
`incr = do n ← `get; `put (n + 1)

incr-test : un ((given hSt handle `incr) 0) ≡ (tt , 1)
incr-test = refl
```

Fig. 1. A state effect (upper), its handler (lower left), and a simple test (lower right) which uses the (elided) smart constructors for get and put

by the CatchM record below. The record asserts that the computation type $M : \mathsf{Set} \to \mathsf{Set}$ has at least a higher-order operation catch and a first-order operation throw:

```
record CatchM (M : Set → Set) : Set₁ where
  field catch : M A → M A → M A
        throw :             M A
```

The idea is that throw throws an exception, and catch m_1 m_2 handles any exception thrown during evaluation of m_1 by running m_2 instead. The problem is that we cannot give a modular definition of operations such as catch using algebraic effects and handlers alone. As discussed in § 1.2, the crux of the problem is that algebraic effects and handlers provide limited support for higher-order operations. However, as also discussed in § 1.2, we can encode catch in terms of more primitive effects and handlers, such as the following handler for the Throw effect:

```
hThrow : ⟨ A ! Throw ⇒ ⊤ ⇒ (Maybe A) ! Δ′ ⟩
ret  hThrow x _ = pure (just x)
hdl hThrow throw k _ = pure nothing
```

The handler modifies the return type of the computation by decorating it with a Maybe. If no exception is thrown, ret wraps the yielded value in a just. If an exception is thrown, the handler never invokes the continuation k and aborts the computation by returning nothing instead. We can elaborate catch into an inline application of hThrow. To do so we make use of *effect masking* which lets us "weaken" the type of a computation by inserting extra effects in an effect row:

```
♯_ : ⦃ Δ ∼ Δ₀ ▸ Δ′ ⦄ → Free Δ′ A → Free Δ A
```

Using this, the following elaboration defines a semantics for the catch operation:[17]

```
catch : ⦃ w : Δ ∼ Throw ▸ Δ′ ⦄ → Free Δ A → Free Δ A → Free Δ A
catch m₁ m₂ = (♯ ((given hThrow handle m₁) tt)) >>= (maybe pure m₂)
```

If m_1 does not throw an exception, we return the produced value. If it does, m_2 is run.

As observed by Wu et al. [2014], programs that use elaborations such as catch are less modular than programs that only use plain algebraic operations. In particular, the effect row type of computations no longer represents the interface of operations that we use to write programs, since

[17] The maybe function is the eliminator for the Maybe type. Its first parameter is for eliminating a just; the second nothing. Its type is maybe : $(A \to B) \to B \to \mathsf{Maybe}\ A \to B$.

the catch elaboration is not represented in the effect type at all. So we have to rely on different machinery if we want to refactor, optimize, or change the semantics of catch without having to change programs that use it. In the next subsection we describe how to define effectful operations such as catch modularly using scoped effects and handlers, and discuss how this is not possible for, e.g., operations representing λ abstraction.

2.6 Scoped Effects and Handlers

This subsection gives an overview of scoped effects and handlers. While the rest of the paper can be read and understood without a deep understanding of scoped effects and handlers, we include this overview to facilitate comparison with the alternative solution that we introduce in § 3.

Scoped effects extend the expressiveness of algebraic effects to support a class of higher-order operations that Piróg et al. [2018]; Wu et al. [2014]; Yang et al. [2022] call *scoped operations*. We illustrate how scoped effects work, using a freer monad encoding of the endofunctor algebra approach of Yang et al. [2022]. The work of Yang et al. [2022] does not include examples of modular handlers, but the original paper on scoped effects and handlers by Wu et al. [2014] does. We describe an adaptation of the modular handling techniques due to Wu et al. [2014] to the endofunctor algebra approach of Yang et al. [2022].

2.6.1 Scoped Programs. Scoped effects extend the free monad data type with an additional row for scoped operations. The return and call constructors of Prog below correspond to the pure and impure constructors of the free monad, whereas enter is new:

```
data Prog (Δ γ : Effect) (A : Set) : Set₁ where
  return : A                                                          → Prog Δ γ A
  call   : (op : Op Δ)                                (k : Ret Δ op → Prog Δ γ A) → Prog Δ γ A
  enter  : (op : Op γ) (sc : Ret γ op → Prog Δ γ B) (k : B          → Prog Δ γ A) → Prog Δ γ A
```

The enter constructor represents a higher-order operation which has as many sub-scopes (i.e., computation parameters) as there are inhabitants of the return type of the operation $(op : \mathsf{Op}\ \gamma)$. Each sub-scope of enter is a *scope* in the sense that control flows from the scope to the continuation, since the return type of each scope (B) matches the parameter type of the continuation k of enter.

Using Prog, the catch operation can be defined as a scoped operation:

```
data CatchOp : Set where
  catch : CatchOp
```

```
Catch : Effect
Op  Catch = CatchOp
Ret Catch catch = Bool
```

The effect signature indicates that Catch has two scopes since Bool has two inhabitants. Following Yang et al. [2022], scoped operations are handled using a structure-preserving fold over Prog:

```
hcata :  (∀ {X} → X → G X)
      → CallAlg  Δ G
      → EnterAlg γ  G
      → Prog Δ γ A → G A
```

```
CallAlg : (Δ : Effect) (G : Set → Set₁) → Set₁
CallAlg Δ G =
  {A : Set} (op : Op Δ) (k : Ret Δ op → G A) → G A

EnterAlg : (γ : Effect) (G : Set → Set₁) → Set₁
EnterAlg γ G =
  {A B : Set} (op : Op γ) (sc : Ret γ op → G B) (k : B → G A)
  → G A
```

The first argument represents the case where we are folding a return node; the second and third correspond to respectively call and enter.

2.6.2 Scoped Effect Handlers. The following defines a type of parameterized scoped effect handlers:

```
record ⟨•!_!_⇒_⇒_•!_!_⟩ (Δ γ : Effect) (P : Set) (G : Set → Set) (Δ′ γ′ : Effect) : Set₁ where
  field ret    : X → P → Prog Δ′ γ′ (G X)
        hcall  : CallAlg  Δ (λ X → P → Prog Δ′ γ′ (G X))
        henter : EnterAlg γ (λ X → P → Prog Δ′ γ′ (G X))
        glue   : (k : C → P → Prog Δ′ γ′ (G X)) (r : G C) → P → Prog Δ′ γ′ (G X)
```

A handler of type $\langle \bullet \,!\, \Delta \,!\, \gamma \Rightarrow P \Rightarrow G \bullet! \,\Delta' \,!\, \gamma' \rangle$ handles operations of Δ and γ *simultaneously* and turns a computation Prog $\Delta\ \gamma\ A$ into a parameterized computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ A)$. The ret and hcall cases are similar to the ret and hdl cases from § 2.4. The crucial addition which adds support for higher-order operations is the henter case which allows handler cases to first invoke scoped sub-computations and inspect their return types, before (optionally) passing control to the continuation k. The glue function is used for modularly *weaving* [Wu et al. 2014] side effects of handlers through sub-scopes of yet-unhandled operations.

2.6.3 Weaving. To see why glue is needed, it is instructional to look at how the fields in the record type above are used to fold over Prog:

```
given_handle-scoped_ :  ⦃ w₁ : Δ ~ Δ₀ ▸ Δ′ ⦄ ⦃ w₂ : γ ~ γ₀ ▸ γ′ ⦄
                        → ⟨•! Δ₀ ! γ₀ ⇒ P ⇒ G •! Δ′ ! γ′ ⟩
                        → Prog Δ γ A → P → Prog Δ′ γ′ (G A)
given h handle-scoped m = hcata
  (ret h)
  [ hcall h , (λ op k p → call op (λ x → k x p)) ]
  [ henter h
  , (λ op sc k p → enter op (λ x → sc x p) (λ x → glue h k x p)) ]
  (to-frontΔ (to-frontγ m))
```

The second to last line above shows how glue is used. Because hcata eagerly folds the current handler over scopes (sc), there is a mismatch between the type that the continuation expects (B) and the type that the scoped computation returns ($G\ B$). The glue function fixes this mismatch for the particular return type modification G : Set → Set of a parameterized scoped effect handler.

The scoped effect handler for exception catching is thus:[18]

```
hCatch : ⟨•! Throw ! Catch ⇒ ⊤ ⇒ Maybe •! Δ′ ! γ′ ⟩
ret    hCatch x _ = return (just x)
hcall  hCatch throw k _ = return nothing
henter hCatch catch sc k p = let m₁ = sc true p; m₂ = sc false p; k = flip k p in
  m₁ >>= maybe k (m₂ >>= maybe k (return nothing))
glue   hCatch k x p = maybe (flip k p) (return nothing) x
```

The henter field for the catch operation first runs m_1. If no exception is thrown, the value produced by m_1 is forwarded to k. Otherwise, m_2 is run and its value is forwarded to k, or its exception is propagated. The glue field of hCatch says that, if an unhandled exception is thrown during evaluation of a scope, the continuation is discarded and the exception is propagated; and if no exception is thrown the continuation proceeds normally.

[18] Here, flip : $(A \to B \to C) \to (B \to A \to C)$.

2.6.4 Discussion and Limitations. As observed by van den Berg et al. [2021], some higher-order effects do not correspond to scoped operations. In particular, the LambdaM record shown below § 2.5 is not a scoped operation:

```
record LambdaM (V : Set) (M : Set → Set) : Set₁ where
  field lam : (V → M V) → M V
        app : V → M V   → M V
```

The lam field represents an operation that constructs a λ value. The app field represents an operation that will apply the function value in the first parameter position to the argument computation in the second parameter position. The app operation has a computation as its second parameter so that it remains compatible with different evaluation strategies.

To see why the operations summarized by the LambdaM record above are not scoped operations, let us revisit the definition of scoped operations, explicating an implicit quantification in the enter constructor of Prog:

```
enter : {B : Set} (op : Op γ) (sc : Ret γ op → Prog Δ γ B) (k : B → Prog Δ γ A) → Prog Δ γ A
```

The highlighted B is *existentially quantified*, meaning that the continuation expects as input a value of some type B that only reveals itself once we match on enter. Consequently, the only way to get a value of this type B is by running the scoped computation *sc*. At the same time, the only thing we can do with the result of running *sc*, is applying it to the continuation, making it impossible to postpone the evaluation of a scoped computation. But that is exactly what the implementation of the lam operation of LambdaM requires. Consequently the lam operation is not a scoped operation. It is possible to elaborate the LambdaM operations into more primitive effects and handlers, but as discussed in §§ 1.2 and 2.5, such elaborations are not modular.

In the next section we present a simple alternative solution to scoped effects which supports a broader class of higher-order effects.

3 HEFTY TREES AND ALGEBRAS

As observed in § 2.5, operations such as catch can be elaborated into more primitive effects and handlers. However, these elaborations are not modular. We propose to solve this problem by factoring these elaborations into interfaces of their own to make them modular.

To this end, we first introduce a new type of abstract syntax trees (§§ 3.1 to 3.3) representing computations with higher-order operations, which we dub *hefty trees* (an acronymic pun on *h*igher-order *ef*fec*t*s). We then define elaborations as algebras (*hefty algebras*; § 3.4) over these trees. The following pipeline summarizes the idea, where H is a *higher-order effect signature*:

$$\mathsf{Hefty}\ H\ A \xrightarrow{\textit{elaborate}} \mathsf{Free}\ \Delta\ A \xrightarrow{\textit{handle}} \textit{Result}$$

For the categorically inclined reader, Hefty conceptually corresponds to the initial algebra of the functor $\mathit{HeftyF}\ H\ A\ R = A + H\ R\ (R\ A)$ where $H : (\mathsf{Set} \to \mathsf{Set}) \to (\mathsf{Set} \to \mathsf{Set})$ defines the signature of higher-order operations and is a *higher-order functor*, meaning we have both the usual functorial $\mathit{map} : (X \to Y) \to H\ F\ X \to H\ F\ Y$ for any functor F as well as a function $\mathit{hmap} : \mathrm{Nat}(F, G) \to \mathrm{Nat}(H\ F, H\ G)$ which lifts natural transformations between any F and G to a natural transformation between $H\ F$ and $H\ G$. A hefty algebra is then an F-algebra over a higher-order signature functor H. The notion of elaboration that we introduce in § 3.4 is an F-algebra whose carrier is a "first-order" effect tree (Free Δ).

In this section, we encode this conceptual framework in Agda using *containers* [Abbott et al. 2003, 2005], which correspond to a higher-order signature functor H by requiring that computation types

only occur in *strictly positive* positions. This allows us to shallowly embed the conceptual framework in Agda, but may be more restrictive than strictly necessary. We discuss further limitations of the approach and compare with previous work in § 3.5.

3.1 Generalizing Free to Support Higher-Order Operations

As summarized in § 2.1, Free Δ A is the type of abstract syntax trees representing computations over the effect signature Δ. Our objective is to arrive at a more general type of abstract syntax trees representing computations involving (possibly) higher-order operations. To realize this objective, let us consider how to syntactically represent this variant of the *censor* operation (§ 1.2), where M is the type of abstract syntax trees whose type we wish to define:

$$\mathsf{censor}_{op} : (\mathsf{String} \to \mathsf{String}) \to M\ \top \to M\ \top$$

We call the second parameter of this operation a *computation parameter*. Using Free, computation parameters can only be encoded as continuations; i.e., inside k of the impure constructor:

$$\mathsf{impure} : (op : \mathsf{Op}\ \Delta)\ (k : \mathsf{Ret}\ \Delta\ op \to \mathsf{Free}\ \Delta\ A) \to \mathsf{Free}\ \Delta\ A$$

But the computation parameter of censor_{op} is *not* a continuation, since

$$\mathbf{do}\ (\mathsf{censor}_{op}\ f\ m);\ \text{`out}\ s \not\equiv \mathsf{censor}_{op}\ f\ (\mathbf{do}\ m;\ \text{`out}\ s).$$

As a first attempt at generalizing Free, we might define a type of abstract syntax trees where *all* operations have a computation parameter. The Effect_1 signature type (left) represents an effect signature for this case, where ParRet defines the return type of the computation parameter of each operation. The syntax tree type on the right defines a type of abstract syntax trees where each operation has exactly one computation parameter (ψ):

```
record Effect₁ : Set₁ where
  field Op    : Set
        ParRet : Op → Set
        Ret    : Op → Set
```

```
data Hefty₁ (H : Effect₁) (A : Set) : Set where
  pure₁   : A → Hefty₁ H A
  impure₁ : (op : Op H)
            (ψ  : Hefty₁ H (ParRet H op))
            (k  : Ret H op → Hefty₁ H A)
          → Hefty₁ H A
```

However, algebraic operations generally do not have any computation parameters, and many higher-order operations have more than one (e.g., the catch operation discussed in § 2.5). For this reason, we further generalize effect signatures to also define how many computation parameters a computation has: the ParAr of the Effect_2 signature below (left) is a type that represents the computation parameter arity of each operation. The abstract syntax tree type (right) defines abstract syntax trees that have as many branches as ParAr has constructors:

```
record Effect₂ : Set₁ where
  field Op    : Set
        ParAr  : Op → Set
        ParRet : Op → Set
        Ret    : Op → Set
```

```
data Hefty₂ (H : Effect₂) (A : Set) : Set where
  pure₂   : A → Hefty₂ H A
  impure₂ : (op : Op H)
            (ψ  : ParAr H op → Hefty₂ H (ParRet H op))
            (k  : Ret H op → Hefty₂ H A)
          → Hefty₂ H A
```

We can now use Effect_2 and Hefty_2 to define the syntax of operations with computation parameters, such as catch and censor_{op}. However, the Effect_2 signature restricts all computation parameters to have the *same* return type. This unnecessarily precludes some higher-order operations, such as as a more general operation for exception catching $\text{`catch-gen} : M\ A \to M\ B \to M\ (A \uplus B)$ which

```
data CensorOp : Set where
  censor : (String → String) → CensorOp

Censor : Effect^H
Op^H Censor = CensorOp
Fork Censor (censor f) = record
  { Op = ⊤ ; Ret = λ _ → ⊤ }
Ret^H Censor (censor s) = ⊤

censor_op : (String → String) → Hefty (Censor + H) ⊤ → Hefty (Censor + H) ⊤
censor_op f m = impure (inj_1 (censor f)) (λ _ → m) pure
```

Fig. 2. A higher-order censor effect and operation, with a single computation parameter (declared with Op = ⊤ in the effect signature top right) with return type ⊤ (declared with Ret = λ _ → ⊤ top right)

returns either A or B, depending on whether the first computation parameter throws an exception at run time. As a last generalization, we therefore allow each computation parameter to have a different return type. We realize this generalization by making the return type of each computation depend on ParAr in the Effect_3 type below, such that the return type of computation parameters varies depending on which computation parameter arity constructor (ParAr) it is given:

```
record Effect₃ : Set₁ where
  field
    Op     : Set
    ParAr  : Op → Set
    ParRet : (op : Op)
           → ParAr op → Set
    Ret    : Op → Set

data Hefty₃ (H : Effect₃) (A : Set) : Set where
  pure₃    : A → Hefty₃ H A
  impure₃ : (op : Op H)
            (ψ  : (a : ParAr H op) → Hefty₃ H (ParRet H op a))
            (k  : Ret H op → Hefty₃ H A)
          → Hefty₃ H A
```

Notice that ParAr and ParRet is actually a signature in disguise. In other words, Effect_3 and Hefty_3 are equivalent to the following notion of *higher-order effect signature* (H : Effect^H) and abstract syntax trees over these:

```
record Effect^H : Set₁ where
  field Op^H  : Set
        Fork  : Op^H → Effect
        Ret^H : Op^H → Set

data Hefty (H : Effect^H) (A : Set) : Set where
  pure    : A → Hefty H A
  impure : (op : Op^H H)
           (ψ : (a : Op (Fork H op)) → Hefty H (Ret (Fork H op) a))
           (k : Ret^H H op → Hefty H A)
         → Hefty H A
```

This type of Hefty trees can be used to define higher-order operations with an arbitrary number of computation parameters, with arbitrary return types. Using this type, and using a co-product for higher-order effect signatures (_+_) which is analogous to the co-product for algebraic effect signatures in § 2.2, Figure 2 represents the syntax of the censor_{op} operation.

Just like Free, Hefty trees can be sequenced using monadic bind. Unlike for Free, the monadic bind of Hefty is not expressible in terms of the standard fold over Hefty trees. The difference between Free and Hefty is that Free is a regular data type whereas Hefty is a *nested datatype* [Bird and Paterson 1999]. The fold of a nested data type is limited to describe *natural transformations*. As Bird and Paterson [1999] show, this limitation can be overcome by using a *generalized fold*, but for the purpose of this paper it suffices to define monadic bind as a recursive function:

```
_>>=_ : Hefty H A → (A → Hefty H B) → Hefty H B
pure x          >>= g = g x
impure op ψ k >>= g = impure op ψ (λ x → k x >>= g)
```

The bind behaves similarly to the bind for Free; i.e., $m \gg\!= g$ concatenates g to all the leaves in the continuations (but not computation parameters) of m.

In § 3.4 we show how to modularly elaborate higher-order operations into more primitive algebraic effects and handlers (i.e., computations over Free), by folding modular elaboration algebras (*hefty algebras*) over Hefty trees. First, we show (in § 3.2) how Hefty trees support programming against an interface of both algebraic and higher-order operations. We also address (in § 3.3) the question of how to encode effect signatures for higher-order operations whose computation parameters have polymorphic return types, such as the highlighted A below:

```
`catch : Hefty H A → Hefty H A → Hefty H A
```

3.2 Programs with Algebraic and Higher-Order Effects

Any algebraic effect signature can be lifted to a higher-order effect signature with no fork (i.e., no computation parameters):

```
Lift : Effect → Effect^H
Op^H  (Lift Δ)   = Op Δ
Fork  (Lift Δ) _ = Nil
Ret^H (Lift Δ)   = Ret Δ
```

Using this effect signature, and using higher-order effect row insertion witnesses analogous to the ones we defined and used in § 2.2, the following smart constructor lets us represent any algebraic operation as a Hefty computation:

```
↑_ : ⦃ w : H ~ Lift Δ ▷ H' ⦄ → (op : Op Δ) → Hefty H (Ret Δ op)
```

Using this notion of lifting, Hefty trees can be used to program against interfaces of both higher-order and plain algebraic effects.

3.3 Higher-Order Operations with Polymorphic Return Types

Let us consider how to define Catch as a higher-order effect. Ideally, we would define an operation that is parameterized by a return type of the branches of a particular catch operation, as shown on the left, such that we can define the higher-order effect signature on the right:[19]

```
data CatchOp^d : Set₁ where
  catch^d : Set → CatchOp^d
```

```
Catch^d : Effect^H
Op^H  Catch^d = CatchOp^d
Fork  Catch^d (catch^d A) = record
  { Op = Bool; Ret = λ _ → A }
Ret^H Catch^d (catch^d A) = A
```

The Fork field on the right says that Catch has two sub-computations (since Bool has two constructors), and that each computation parameter has some return type A. However, the signature on the right above is not well defined!

The problem is that, because CatchOp^d has a constructor that quantifies over a type (Set), the CatchOp^d type lives in Set_1. Consequently it does not fit the definition of Effect^H, whose operations live in Set. There are two potential solutions to this problem: (1) increase the universe level of

[19] d is for *dubious*.

Effect^H to allow Op^H to live in Set_1; or (2) use a *universe of types* [Martin-Löf 1984]. Either solution is applicable here. However, for some operations (e.g., λ in § 5.1) it is natural to model types as an interface that we are programming against. For this reason, using a type universe is a natural fit.

A universe of types is a (dependent) pair of a syntax of types ($\mathsf{Ty} : \mathsf{Set}$) and a semantic function ($[\![\_]\!] : \mathsf{Ty} \to \mathsf{Set}$) defining the meaning of the syntax by reflecting it into Agda's Set:

```
record Universe : Set₁ where
  field Ty   : Set
        ⟦_⟧ : Ty → Set
```

Using type universes, we can parameterize the catch constructor on the left below by a *syntactic type* Ty of some universe u, and use the *meaning of this type* ($[\![\ t\ ]\!]$) as the return type of the computation parameters in the effect signature on the right below:

```
data CatchOp ⦃ u : Universe ⦄ : Set where
  catch : Ty → CatchOp
```

```
Catch : ⦃ u : Universe ⦄ → Effect^H
Op^H  Catch = CatchOp
Fork  Catch (catch t) = record
  { Op = Bool; Ret = λ _ → ⟦ t ⟧ }
Ret^H Catch (catch t) = ⟦ t ⟧
```

While the universe of types encoding restricts the kind of type that catch can have as a return type, the effect signature is parametric in the universe. Thus the implementer of the Catch effect signature (or interface) is free to choose a sufficiently expressive universe of types.

3.4 Hefty Algebras

As shown in § 2.5, the higher-order catch operation can be encoded as a non-modular elaboration:

```
catch m₁ m₂ = (♯ ((given hThrow handle m₁) tt)) >>= (maybe pure m₂)
```

We can make this elaboration modular by expressing it as an *algebra* over Hefty trees containing operations of the Catch signature. To this end, we will use the following notion of hefty algebra (Alg^H) and fold (or *catamorphism* [Meijer et al. 1991], cata^H) for Hefty:

```
record Alg^H (H : Effect^H) (F : Set → Set) : Set₁ where
  field alg : (op : Op^H H)
              (ψ  : (s : Op (Fork H op)) → F (Ret (Fork H op) s))
              (k  : Ret^H H op → F A)
              → F A

cata^H : (∀ {A} → A → F A) → Alg^H H F → Hefty H A → F A
cata^H g a (pure x)          = g x
cata^H g a (impure op ψ k) = alg a op (cata^H g a ∘ ψ) (cata^H g a ∘ k)
```

Here Alg^H defines how to transform an impure node of type Hefty $H\ A$ into a value of type $F\ A$, assuming we have already folded the computation parameters and continuation into F values. A nice property of algebras is that they are closed under higher-order effect signature sums:

```
_∨_ : Alg^H H₁ F → Alg^H H₂ F → Alg^H (H₁ + H₂) F
alg (A₁ ∨ A₂) = [ alg A₁ , alg A₂ ]
```

By defining elaborations as hefty algebras (below) we can compose them using _∨_.

```
Elaboration : Effect^H → Effect → Set₁
Elaboration H Δ = Alg^H H (Free Δ)
```

An Elaboration $H\ \Delta$ elaborates higher-order operations of signature H into algebraic operations of signature Δ. Given an elaboration, we can generically transform any hefty tree into more primitive algebraic effects and handlers:

```
elaborate : Elaboration H Δ → Hefty H A → Free Δ A
elaborate = cata^H pure
```

Example 3.1. The elaboration below is analogous to the non-modular catch elaboration:

```
eCatch : ⦃ u : Universe ⦄ ⦃ w : Δ ~ Throw ▸ Δ′ ⦄ → Elaboration Catch Δ
alg eCatch (catch t) ψ k = let m₁ = ψ true; m₂ = ψ false in
  (♯ ((given hThrow handle m₁) tt)) >>= maybe k (m₂ >>= k)
```

The elaboration is essentially the same as its non-modular counterpart, except that it now uses the universe of types encoding discussed in § 3.3, and that it now transforms syntactic representations of higher-order operations instead. Using this elaboration, we can, for example, run the following example program involving the state effect from Figure 1, the throw effect from § 2.1, and the catch effect defined here:

```
transact :  ⦃ w_s : H ~ Lift State ▸ H′ ⦄ ⦃ w_t : H ~ Lift Throw ▸ H″ ⦄ ⦃ w : H ~ Catch ▸ H‴ ⦄
         → Hefty H ℕ
transact = do
  ↑ (put 1)
  `catch (do ↑ (put 2); (↑ throw) >>= ⊥-elim) (pure tt)
  ↑ get
```

The program first sets the state to 1; then to 2; and then throws an exception. The exception is caught, and control is immediately passed to the final operation in the program which gets the state. By also defining elaborations for Lift and Nil, we can elaborate and run the program:

```
eTransact : Elaboration (Catch + Lift Throw + Lift State + Lift Nil) (Throw ⊕ State ⊕ Nil)
eTransact = eCatch ∨ eLift ∨ eLift ∨ eNil

test-transact : un ( ( given hSt
                       handle ( ( given hThrow
                                  handle (elaborate eTransact transact)))
                                tt ) 0 ) ≡ (just 2 , 2)
test-transact = refl
```

The program above uses a so-called *global* interpretation of state, where the put operation in the "try block" of `catch causes the state to be updated globally. In § 5.2.2 we return to this example and show how we can modularly change the elaboration of the higher-order effect Catch to yield a so-called *transactional* interpretation of state where the put operation in the try block is rolled back when an exception is thrown.

3.5 Discussion and Limitations

Which (higher-order) effects can we describe using hefty trees and algebras? Since the core mechanism of our approach is modular elaboration of higher-order operations into more primitive effects and handlers, it is clear that hefty trees and algebras are at least as expressive as standard algebraic effects. The crucial benefit of hefty algebras over algebraic effects is that higher-order operations can be declared and implemented modularly. In this sense, hefty algebras provide a

modular abstraction layer over standard algebraic effects that, although it adds an extra layer of indirection by requiring both elaborations and handlers to give a semantics to hefty trees, is comparatively cheap and implemented using only standard techniques such as F-algebras.

Conceptually, we expect that hefty trees can capture any *monadic* higher-order effect whose signature is given by a higher-order functor on Set → Set. Filinski [1999] showed that any monadic effect can be represented using continuations, and given that we can encode the continuation monad using algebraic effects [Schrijvers et al. 2019] in terms of the *sub/jump* operations (§ 5.2.2) by Fiore and Staton [2014]; Thielecke [1997], it is possible to elaborate any monadic effect into algebraic effects using hefty algebras. The current Agda implementation, though, is slightly more restrictive. The type of effect signatures, Effect^H, approximates the set of higher-order functors by constructively enforcing that all occurrences of the computation type are strictly positive. Hence, there may be higher-order effects that are well-defined semantically, but which cannot be captured in the Agda encoding presented here.

When comparing hefty trees to scoped effects, we observe two important differences. The first difference is that the syntax of programs with higher-order effects is fundamentally more restrictive when using scoped effects. Specifically, as discussed at the end of § 2.6.4, scoped effects impose a restriction on operations that their computation parameters must pass control directly to the continuation of the operation. Hefty trees, on the other hand, do not restrict the control flow of computation parameters, meaning that they can be used to define a broader class of operations. For instance, in § 5.1 we define a higher-order effect for function abstraction, which is an example of an operation where control does not flow from the computation parameter to the continuation.

The second difference is that hefty algebras and scoped effects and handlers are modular in different ways. Scoped effects are modular because we can modularly define, compose, and handle scoped operations, by applying scoped effect handlers in sequence; i.e.:

$$\mathsf{Prog}\ \Delta_0\ \gamma_0\ A_0 \xrightarrow{h_1} \mathsf{Prog}\ \Delta_1\ \gamma_1\ A_1 \xrightarrow{h_2} \cdots \xrightarrow{h_n} \mathsf{Prog}\ \mathsf{Nil}\ \mathsf{Nil}\ A_n$$

As discussed in § 2.6.3, each handler application modularly "weaves" effects through sub computations, using a dedicated glue function. Hefty algebras, on the other hand, work by applying an elaboration algebra assembled from modular components in one go. The program resulting from elaboration can then be handled using standard algebraic effect handlers; i.e.:

$$\mathsf{Hefty}\ (H_0 \dotplus \cdots \dotplus H_m)\ A \xrightarrow{\text{elaborate } (E_0 \vee \cdots \vee E_m)} \mathsf{Free}\ \Delta\ A \xrightarrow{h_1} \cdots \xrightarrow{h_k} \mathsf{Free}\ \mathsf{Nil}\ A$$

Because hefty algebras eagerly elaborate all higher-order effects in one go, they do not require similar "weaving" as scoped effect handlers. A consequence of this difference is that scoped effect handlers exhibit more effect interaction by default; i.e., different permutations of handlers may give different semantics. In contrast, when using hefty algebras we have to be more explicit about such effect interactions. We discuss this difference in more detail in § 5.2.2.

4 VERIFYING ALGEBRAIC LAWS FOR HIGHER-ORDER EFFECTS

A key idea behind algebraic effects is that we can state and prove algebraic laws about effectful operations. In this section we show how to verify the lawfulness of catch , and compare the effort required to verify lawfulness using hefty algebras vs. a non-modular elaboration for catch.

The record type shown below defines the interface of a monad given by the record parameters M, *return*, and _≫=_. The fields on the left below assert that M has a *throw* and *catch* operation, as well as a run function which runs a computation to yield a result R : Set → Set.[20] On the right are

[20]The notation ⦃ u ⦄ : Universe treats the u field as an *instance* that can be automatically resolved in the scope of the CatchIntf record type.

```
u            (CatchImpl₀ ⦃ u ⦄) = u
throw        CatchImpl₀         = `throwᴴ
catch        CatchImpl₀         = `catch
R            CatchImpl₀         = Free Δ ∘ Maybe
run          CatchImpl₀         = h ∘ e

bind-throw   CatchImpl₀ k   = refl
catch-return CatchImpl₀ x m = refl
catch-throw₁ CatchImpl₀ m   = begin
    h (e (`catch `throwᴴ m))
  ≡⟨ refl ⟩
    h ((♯ h (e `throwᴴ)) >>= maybe pure ((e m) >>= pure))
  ≡⟨ cong! (Free-unitᵣ-≡ (e m)) ⟩
    h (e m)   □
catch-throw₂ CatchImpl₀ m   = begin
    h (e (`catch m `throwᴴ))
  ≡⟨ refl ⟩
    h ((♯ h (e m)) >>= maybe pure ((e `throwᴴ) >>= pure))
  ≡⟨ cong (λ P → h ((♯ h (e m)) >>= P))
       (extensionality (λ x →
         cong (λ P → maybe pure P x)
           (cong (impure (inj₁ throw))
                 (extensionality (λ x → ⊥-elim x))))) ⟩
    h ((♯ h (e m)) >>= maybe pure `throw)
  ≡⟨ catch-throw-lem (e m) ⟩
    h (e m)   □
```

```
u            (CatchImpl₁ ⦃ u ⦄) = u
throw        CatchImpl₁         = `throw
catch        CatchImpl₁         = catch
R            CatchImpl₁         = Free Δ ∘ Maybe
run          CatchImpl₁         = h

bind-throw   CatchImpl₁ k   = refl
catch-return CatchImpl₁ x m = refl
catch-throw₁ CatchImpl₁ m   = refl

catch-throw₂ CatchImpl₁ m   = begin
    h (catch m `throw)
  ≡⟨ refl ⟩
    h ((♯ h m) >>= maybe pure `throw)
  ≡⟨ catch-throw-lem m ⟩
    h m   □
```

Fig. 3. Lawfulness for the modular elaboration (left) and the non-modular elaboration of catch (right)

the laws that constrain the behavior of the throw and catch operations. The laws are borrowed from Delaware et al. [2013].

```
record CatchIntf (M : Set → Set)
       (return  :  ∀ {A} → A → M A)
       (_>>=_ : ∀ {A B}
              → M A → (A → M B) → M B) : Set₁ where
  field ⦃ u ⦄ : Universe
        throw  : {t : Ty} → M ⟦ t ⟧
        catch  : {t : Ty} → M ⟦ t ⟧ → M ⟦ t ⟧ → M ⟦ t ⟧
        R      : Set → Set
        run    : M A → R A
```

```
bind-throw : {t₁ t₂ : Ty} (k : ⟦ t₁ ⟧ → M ⟦ t₁ ⟧)
  → run (throw >>= k) ≡ run throw
catch-throw₁ : {t : Ty} (m : M ⟦ t ⟧)
  → run (catch throw m) ≡ run m
catch-throw₂ : {t : Ty} (m : M ⟦ t ⟧)
  → run (catch m throw) ≡ run m
catch-return : {t : Ty} (x : ⟦ t ⟧) (m : M ⟦ t ⟧)
  → run (catch (return x) m) ≡ run (return x)
```

Figure 3 (left) shows that the elaboration and handler from the previous section satisfy these laws. The figure uses `throwᴴ as an abbreviation for ↑ throw >>= ⊥-elim, h as an abbreviation of the handler for hThrow, and e as an abbreviation of elaborate. The proofs are equational rewriting proofs akin to pen-and-paper proofs, except that each step is mechanically verified. The equational rewriting steps use the ≡-Reasoning module from the Agda standard library, and have the form $t_1 \equiv\langle\ eq\ \rangle\ t_2$ where t_1 is the term before the rewrite, t_2 is the term after, and eq is a proof that t_1 and t_2 are equal. The question is, how much overhead the hefty algebra encoding adds compared to the non-modular abbreviation of catch from § 2.5? To answer this question, Figure 3 also contains the implementation and proof of a non-modular elaboration of catch (CatchImpl_1 on the right).

The side-by-side comparison shows that hefty algebra elaborations add some administrative overhead. In particular, elaborations introduce some redundant binds, as in the sub-term (e *m*) ≫= pure of the term resulting from the first equational rewrite in catch-throw$_1$ on the left above. These extraneous binds are rewritten away by applying the free monad right unit law (Free-unit$_r$-≡). Another source of overhead of using hefty algebras is that Agda is unable to infer that the term resulting from elaborating `throw$^H$ is equal to the term given by the smart constructor `throw. We prove this explicitly on the left above in the second-to-last equational rewrite of catch-throw$_2$. Both proofs make use of functional extensionality (which is postulated since we cannot prove functional extensionality in general in Agda), and a straightforward catch-throw-lem lemma that we prove by induction on the structure of the computation parameter of the lemma.

Except for the administrative overhead discussed above, the proofs have the same structure, and the effort of verifying algebraic laws for higher-order effects defined using hefty algebras is about the same as verifying algebraic laws for direct, non-modular encodings.

5 EXAMPLES

As discussed in § 2.5, there is a wide range of examples of higher-order effects that cannot be defined as algebraic operations directly, and are typically defined as non-modular elaborations instead. In this section we give examples of such effects and show to define them modularly using hefty algebras. The artifact [Bach Poulsen and Reinders 2023] contains the full examples.

5.1 λ as a Higher-Order Operation

As recently observed by van den Berg et al. [2021], the (higher-order) operations for λ abstraction and application are neither algebraic nor scoped effects. We demonstrate how hefty algebras allow us to modularly define and elaborate an interface of higher-order operations for λ abstraction and application, inspired by Levy's call-by-push-value [Levy 2006]. The interface we will consider is parametric in a universe of types given by the following record:

```
record LamUniverse : Set₁ where
  field ⦃ u ⦄ : Universe
        _↣_ : Ty → Ty → Ty
        c   : Ty → Ty
```

The _↣_ field represents a function type, whereas c is the type of *thunk values*. Distinguishing thunks in this way allows us to assign either a call-by-value or call-by-name semantics to the interface for λ abstraction summarized by the following smart constructors:

```
`lam : {t₁ t₂ : Ty} → (⟦ c t₁ ⟧ → Hefty H ⟦ t₂ ⟧)  → Hefty H ⟦ (c t₁) ↣ t₂ ⟧
`var : {t : Ty}     → ⟦ c t ⟧                     → Hefty H ⟦ t ⟧
`app : {t₁ t₂ : Ty} → ⟦ (c t₁) ↣ t₂ ⟧ → Hefty H ⟦ t₁ ⟧ → Hefty H ⟦ t₂ ⟧
```

Here `lam is a higher-order operation with a function typed computation parameter and whose return type is a function value (⟦ c $t_1$ ↣ $t_2$ ⟧). The `var operation accepts a thunk value as argument and yields a value of a matching type. The `app operation is also a higher-order operation: its first parameter is a function value type, whereas its second parameter is a computation parameter whose return type matches the function value parameter type.

The interface above defines a kind of *higher-order abstract syntax* [Pfenning and Elliott 1988] which piggy-backs on Agda functions for name binding. However, unlike most Agda functions, the constructors above represent functions with side-effects. The representation in principle supports functions with arbitrary side-effects since it is parametric in what the higher-order effect signature H is. Furthermore, we can assign different operational interpretations to the operations in the

interface without having to change the interface or programs written against the interface. To illustrate we give two different implementations of the interface: one that implements a call-by-value evaluation strategy, and one that implements call-by-name.

5.1.1 Call-by-Value. We give a call-by-value interpretation `'lam` by generically elaborating to algebraic effect trees with any set of effects Δ. Our interpretation is parametric in proof witnesses that the following isomorphisms hold for value types ($\leftrightarrow$ is the type of isomorphisms from the Agda standard library):[21]

```
iso₁ : {t₁ t₂ : Ty} → ⟦ t₁ ↣ t₂ ⟧ ↔ (⟦ t₁ ⟧ → Free Δ ⟦ t₂ ⟧)
iso₂ : {t : Ty}     → ⟦ c t      ⟧ ↔ ⟦ t ⟧
```

The first isomorphism says that a function value type corresponds to a function which accepts a value of type t_1 and produces a computation whose return type matches the function type. The second says that thunk types coincide with value types. Using these isomorphisms, the following defines a call-by-value elaboration of functions:

```
eLamCBV : Elaboration Lam Δ
alg eLamCBV lam      ψ k = k (from ψ)
alg eLamCBV (var x)  _ k = k (to x)
alg eLamCBV (app f) ψ k = do
  a ← ψ tt
  v ← to f (from a)
  k v
```

The `lam` case passes the function body given by the sub-tree ψ as a value to the continuation, where the `from` function mediates the sub-tree of type $[\![\ c\ t_1\ ]\!] \to \mathsf{Free}\ \Delta\ [\![\ t_2\ ]\!]$ to a value type $[\![\ (c\ t_1) \rightarrowtail t_2\ ]\!]$, using the isomorphism iso_1. The `var` case uses the `to` function to mediate a $[\![\ c\ t\ ]\!]$ value to a $[\![\ t\ ]\!]$ value, using the isomorphism iso_2. The `app` case first eagerly evaluates the argument expression of the application (in the sub-tree ψ) to an argument value, and then passes the resulting value to the function value of the application. The resulting value is passed to the continuation.

Using the elaboration above, we can evaluate programs such as the following which uses both the higher-order lambda effect, the algebraic state effect, and assumes that our universe has a number type $[\![\ num\ ]\!] \leftrightarrow \mathbb{N}$:

```
ex : Hefty (Lam + Lift State + Lift Nil) ℕ
ex = do
  ↑ put 1
  f ← 'lam (λ x → do
        n₁ ← 'var x
        n₂ ← 'var x
        pure (from ((to n₁) + (to n₂))))
  v ← 'app f incr
  pure (to v)
  where incr = do s₀ ← ↑ get; ↑ put (s₀ + 1); s₁ ← ↑ get; pure (from s₁)
```

The program first sets the state to 1. Then it constructs a function that binds a variable x, dereferences the variable twice, and adds the two resulting values together. Finally, the application in the second-to-last line applies the function with an argument expression which increments the state by 1 and

[21] The two sides of an isomorphism $A \leftrightarrow B$ are given by the functions $\mathsf{to} : A \to B$ and $\mathsf{from} : B \to A$.

returns the resulting value. Running the program produces 4 since the state increment expression is eagerly evaluated before the function is applied.

```
elab-cbv : Elaboration (Lam + Lift State + Lift Nil) (State ⊕ Nil)
elab-cbv = eLamCBV ∨ eLift ∨ eNil

test-ex-cbv : un ((given hSt handle (elaborate elab-cbv ex)) 0) ≡ (4 , 2)
test-ex-cbv = refl
```

5.1.2 Call-by-Name. The key difference between the call-by-value and the call-by-name interpretation of our λ operations is that we now assume that thunks are computations. That is, we assume that the following isomorphisms hold for value types:

```
iso₁ : {t₁ t₂ : Ty} → ⟦ t₁ ↣ t₂ ⟧ ↔ (⟦ t₁ ⟧ → Free Δ ⟦ t₂ ⟧)
iso₂ : {t : Ty}     → ⟦ c t ⟧      ↔ Free Δ ⟦ t ⟧
```

Using these isomorphisms, the following defines a call-by-name elaboration of functions:

```
eLamCBN : Elaboration Lam Δ
alg eLamCBN lam      ψ k = k (from ψ)
alg eLamCBN (var x) _ k = to x >>= k
alg eLamCBN (app f) ψ k = to f (from (ψ tt)) >>= k
```

The case for lam is the same as the call-by-value elaboration. The case for var now needs to force the thunk by running the computation and passing its result to k. The case for app passes the argument sub-tree (ψ) as an argument to the function f, runs the computation resulting from doing so, and then passes its result to k. Running the example program ex from above now produces 5 as result, since the state increment expression in the argument of `app is thunked and run twice during the evaluation of the called function.

```
elab-cbn : Elaboration (Lam + Lift State + Lift Nil) (State ⊕ Nil)
elab-cbn = eLamCBN ∨ eLift ∨ eNil

test-ex-cbn : un ((given hSt handle (elaborate elab-cbn ex)) 0) ≡ (5 , 3)
test-ex-cbn = refl
```

5.2 Optionally Transactional Exception Catching

A feature of scoped effect handlers [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] is that changing the order of handlers makes it possible to obtain different semantics of *effect interaction*. A classical example of effect interaction is the interaction between state and exception catching that we briefly considered at the end of § 3.4 in connection with this transact program:

```
transact : ⦃ wₛ : H ~ Lift State ▹ H′ ⦄ ⦃ wₜ : H ~ Lift Throw ▹ H″ ⦄ ⦃ w : H ~ Catch ▹ H‴ ⦄
         → Hefty H ℕ
transact = do
  ↑ put 1
  `catch (do ↑ put 2; (↑ throw) >>= ⊥-elim) (pure tt)
  ↑ get
```

The state and exception catching effect can interact to give either of these two semantics:

(1) *Global* interpretation of state, where the transact program returns 2 since the put operation in the "try" block causes the state to be updated globally.

(2) *Transactional* interpretation of state, where the transact program returns 1 since the state changes of the put operation are *rolled back* when the "try" block throws an exception.

With monad transformers [Cenciarelli and Moggi 1993; Liang et al. 1995] we can recover either of these semantics by permuting the order of monad transformers. With scoped effect handlers we can also recover either by permuting the order of handlers. However, the eCatch elaboration in § 3.4 always gives us a global interpretation of state. In this section we demonstrate how we can recover a transactional interpretation of state by using a different elaboration of the catch operation into an algebraically effectful program with the throw operation and the off-the-shelf *sub/jump* control effects due to Fiore and Staton [2014]; Thielecke [1997]. The different elaboration is modular in the sense that we do not have to change the interface of the catch operation nor any programs written against the interface.

5.2.1 Sub/Jump. We recall how to define two operations, sub and jump, due to [Fiore and Staton 2014; Thielecke 1997]. We define these operations as algebraic effects following Schrijvers et al. [2019]. The algebraic effects are summarized by the following smart constructors where CC *Ref* is associated with the sub/jump operations:

$$\texttt{`sub} \;:\; \{\!|\, w : \Delta \sim \mathsf{CC}\ \mathit{Ref} \blacktriangleright \Delta' \,|\!\}\ (b : \mathit{Ref}\ t \to \mathsf{Free}\ \Delta\ A)\ (k : [\![\ t\ ]\!] \to \mathsf{Free}\ \Delta\ A) \to \mathsf{Free}\ \Delta\ A$$
$$\texttt{`jump} : \{\!|\, w : \Delta \sim \mathsf{CC}\ \mathit{Ref} \blacktriangleright \Delta' \,|\!\}\ (\mathit{ref} : \mathit{Ref}\ t)\ (x : [\![\ t\ ]\!]) \to \mathsf{Free}\ \Delta\ B$$

An operation `sub $f$ $g$ gives a computation $f$ access to the continuation $g$ via a reference value *Ref* $t$ which represents a continuation expecting a value of type $[\![\ t\ ]\!]$. The `jump operation invokes such continuations. The operations and their handler (abbreviated to h) satisfy the following laws:

$$\mathsf{h}\ (\texttt{`sub}\ (\lambda\ \_ \to p)\ k) \equiv \mathsf{h}\ p$$
$$\mathsf{h}\ (\texttt{`sub}\ (\lambda\ r \to m \gg\!= \texttt{`jump}\ r)\ k) \equiv \mathsf{h}\ (m \gg\!= k)$$
$$\mathsf{h}\ (\texttt{`sub}\ p\ (\texttt{`jump}\ r')) \equiv \mathsf{h}\ (p\ r')$$
$$\mathsf{h}\ (\texttt{`sub}\ p\ q \gg\!= k) \equiv \mathsf{h}\ (\texttt{`sub}\ (\lambda\ x \to p\ x \gg\!= k)\ (\lambda\ x \to q\ x \gg\!= k))$$

The last law asserts that `sub and `jump are *algebraic* operations, since their computational subterms behave as continuations. Thus, we encode `sub and its handler as an algebraic effect.

5.2.2 Optionally Transactional Exception Catching. By using the `sub and `jump operations in our elaboration of catch, we get a semantics of exception catching whose interaction with state depends on the order that the state effect and sub/jump effect is handled.

```
eCatchOT : {| w1 : Δ ~ CC Ref ▸ Δ' |} {| w2 : Δ ~ Throw ▸ Δ'' |} → Elaboration Catch Δ
alg eCatchOT (catch x) ψ k = let m1 = ψ true; m2 = ψ false in
  `sub (λ r → (♯ ((given hThrow handle m1) tt)) >>= maybe k (`jump r (from tt)))
       (λ _ → m2 >>= k)
```

The elaboration uses `sub to capture the continuation of a higher-order catch operation. If no exception is raised, then control flows to the continuation $k$ without invoking the continuation of `sub. Otherwise, we jump to the continuation of `sub, which runs $m_2$ before passing control to $k$. Capturing the continuation in this way interacts with state because the continuation of `sub may have been pre-applied to a state handler that only knows about the "old" state. This happens when we invoke the state handler before the handler for sub/jump: in this case we get the transactional interpretation of state, so running transact gives 1. Otherwise, if we run the sub/jump handler before the state handler, we get the global interpretation of state and the result 2.

The sub/jump elaboration above is more involved than the scoped effect handler that we considered in § 2.6. However, the complicated encoding does not pollute the higher-order effect interface, and only turns up if we strictly want or need effect interaction.

5.3 Logic Programming

Following [Schrijvers et al. 2014; Wu et al. 2014; Yang et al. 2022] we can define a non-deterministic choice operation (_`or_) as an algebraic effect, to provide support for expressing the kind of non-deterministic search for solutions that is common in logic programming. We can also define a `fail operation which indicates that the search in the current branch was unsuccessful. The smart constructors below are the lifted higher-order counterparts to the `or and `fail operations:

```
_`or^H_ : ⦃ H ∼ Lift Choice ▸ H′ ⦄ → Hefty H A → Hefty H A → Hefty H A
`fail^H  : ⦃ H ∼ Lift Choice ▸ H′ ⦄                          → Hefty H A
```

A useful operator for cutting non-deterministic search short when a solution is found is the `once operator. The `once operator is not an algebraic effect, but a scoped (and thus higher-order) effect.

```
`once : ⦃ w : H ∼ Once ▸ H′ ⦄ {t : Ty} → Hefty H ⟦ t ⟧ → Hefty H ⟦ t ⟧
```

We can define the meaning of the once operator as the following elaboration:

```
eOnce : ⦃ Δ ∼ Choice ▸ Δ′ ⦄ → Elaboration Once Δ
alg eOnce once ψ k = do
  l ← ♯ ((given hChoice handle (ψ tt)) tt)
  maybe k `fail (head l)
```

The elaboration runs the branch (ψ) of once under the hChoice handler to compute a list of all solutions of ψ. It then tries to choose the first solution and pass that to the continuation k. If the branch has no solutions, we fail. Under a strict evaluation order, the elaboration computes all possible solutions which is doing more work than needed. Agda 2.6.2.2 does not have a specified evaluation strategy, but does compile to Haskell which is lazy. In Haskell, the solutions would be lazily computed, such that the once operator cuts search short as intended.

5.4 Concurrency

Finally, we consider how to define higher-order operations for concurrency, inspired by Yang et al.'s [2022] *resumption monad* [Claessen 1999; Piróg and Gibbons 2014; Schmidt 1986] definition using scoped effects. We summarize our encoding and compare it with the resumption monad. The goal is to define the following operations:

```
`spawn  : {t : Ty} → (m₁ m₂ : Hefty H ⟦ t ⟧) → Hefty H ⟦ t ⟧
`atomic : {t : Ty} → Hefty H ⟦ t ⟧           → Hefty H ⟦ t ⟧
```

The operation `spawn $m_1$ $m_2$ spawns two threads that run concurrently, and returns the value produced by $m_1$ after both have finished. The operation `atomic m represents a block to be executed atomically; i.e., no other threads run before the block finishes executing.

We elaborate `spawn by interleaving the sub-trees of its computations. To this end, we use a dedicated function which interleaves the operations in two trees and yields as output the value of the left input tree (the first computation parameter):

```
interleave_l :  {Ref : Ty → Set} → Free (CC Ref ⊕ Δ) A → Free (CC Ref ⊕ Δ) B
                → Free (CC Ref ⊕ Δ) A
```

Here, the CC effect is the sub/jump effect that we also used in § 5.2.2. The interleave_l function ensures atomic execution by only interleaving code that is not wrapped in a `sub operation. We

elaborate Concur into CC as follows, where the to-front and from-front functions use the row insertion witness w_a to move the CC effect to the front of the row and back again:

```
eConcur : {Ref : Ty → Set} ⦃ w : Δ ∼ CC Ref ▸ Δ′′ ⦄ → Elaboration Concur Δ
alg eConcur (spawn t) ψ k =
  from-front (interleave_l (to-front (ψ true)) (to-front (ψ false))) >>= k
alg eConcur (atomic t) ψ k = `sub (λ ref → ψ tt >>= `jump ref) k
```

The elaboration uses `sub as a delimiter for blocks that should not be interleaved, such that the $\text{interleave}_l$ function only interleaves code that does not reside in atomic blocks. At the end of an atomic block, we `jump to the (possibly interleaved) computation context, k. By using `sub to explicitly delimit blocks that should not be interleaved, we have encoded what Wu et al. [2014, § 7] call *scoped syntax*.

Example 5.1. Below is an example program that spawns two threads that use the Output effect. The first thread prints 0, 1, and 2; the second prints 3 and 4.

```
ex-01234 : Hefty (Lift Output + Concur + Lift Nil) ℕ
ex-01234 = `spawn (do ↑ out "0"; ↑ out "1"; ↑ out "2"; pure 0)
                  (do ↑ out "3"; ↑ out "4"; pure 0)
```

Since the Concur effect is elaborated to interleave the effects of the two threads, the printed output appears in interleaved order:

```
test-ex-01234 : un ( ( given hOut
                       handle ( ( given hCC
                                    handle (elaborate concur-elab ex-01234)
                                ) tt ) ) tt ) ≡ (0 , "03142")
test-ex-01234 = refl
```

The following program spawns an additional thread with an `atomic block

```
ex-01234567 : Hefty (Lift Output + Concur + Lift Nil) ℕ
ex-01234567 = `spawn ex-01234
                     (`atomic (do ↑ out "5"; ↑ out "6"; ↑ out "7"; pure 0))
```

Inspecting the output, we see that the additional thread indeed computes atomically:

```
test-ex-01234567 : un ( ( given hOut
                          handle ( ( given hCC
                                       handle (elaborate concur-elab ex-01234567)
                                   ) tt ) ) tt ) ≡ (0 , "05673142")
test-ex-01234567 = refl
```

The example above is inspired by the resumption monad, and in particular by the scoped effects definition of concurrency due to Yang et al. [2022]. Yang et al. do not (explicitly) consider how to define the concurrency operations in a modular style. Instead, they give a direct semantics that translates to the resumption monad which we can encode as follows in Agda (assuming resumptions are given by the free monad):

```
data Resumption Δ A : Set where
  done : A                          → Resumption Δ A
  more : Free Δ (Resumption Δ A) → Resumption Δ A
```

We could elaborate into this type using a hefty algebra Alg^H Concur (Resumption Δ) but that would be incompatible with our other elaborations which use the free monad. For that reason, we emulate the resumption monad using the free monad instead of using the Resumption type directly.

6 RELATED WORK

As stated in the introduction of this paper, defining abstractions for programming constructs with side effects is a research question with a long and rich history, which we briefly summarize here. Moggi [1989a] introduced monads as a means of modeling side effects and structuring programs with side effects; an idea which Wadler [1992] helped popularize. A problem with monads is that they do not naturally compose. A range of different solutions have been developed to address this issue [Cenciarelli and Moggi 1993; Filinski 1999; Jones and Duponcheel 1993; Steele Jr. 1994]. Of these solutions, monad transformers [Cenciarelli and Moggi 1993; Jaskelioff 2008; Liang et al. 1995] is the more widely adopted solution. However, more recently, algebraic effects [Plotkin and Power 2002] was proposed as an alternative solution which offers some modularity benefits over monads and monad transformers. In particular, whereas monads and monad transformers may "leak" information about the implementation of operations, algebraic effects enforce a strict separation between the interface and implementation of operations. Furthermore, monad transformers commonly require glue code to "lift" operations between layers of monad transformer stacks. While the latter problem is addressed by the Monatron framework of Jaskelioff [2008], algebraic effects have a simple composition semantics that does not require intricate liftings.

However, some effects, such as exception catching, did not fit into the framework of algebraic effects. *Effect handlers* [Plotkin and Pretnar 2009] were introduced to address this problem. Algebraic effects and handlers has since been gaining traction as a framework for modeling and structuring programs with side effects in a modular way. Several libraries have been developed based on the idea such as *Handlers in Action* [Kammar et al. 2013], the freer monad [Kiselyov and Ishii 2015], or Idris' `Effects` DSL [Brady 2013b]; but also standalone languages such as Eff [Bauer and Pretnar 2015], Koka [Leijen 2017], Frank [Lindley et al. 2017], and Effekt [Brachthäuser et al. 2020].[22]

As discussed in § 1.2 and § 2.5, some modularity benefits of algebraic effects and handlers do not carry over to higher-order effects. Scoped effects and handlers [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] address this shortcoming for *scoped operations*, as we summarized in § 2.6. This paper provides a different solution to the modularity problem with higher-order effects. Our solution is to provide modular elaborations of higher-order effects into more primitive effects and handlers. We can, in theory, encode any effect in terms of algebraic effects and handlers. However, for some effects, the encodings may be complicated. While the complicated encodings are hidden behind a higher-order effect interface, complicated encodings may hinder understanding the operational semantics of higher-order effects, and may make it hard to verify algebraic laws about implementations of the interface. Our framework would also support elaborating higher-order effects into scoped effects and handlers, which might provide benefits for verification. We leave this as a question to explore in future work.

Although not explicitly advertised, some standalone languages, such as Frank [Lindley et al. 2017] and Koka [Leijen 2017] do have some support for higher-order effects. The denotational semantics of these features of these languages is unclear. A question for future work is whether the modular elaborations we introduce could provide a denotational model.

A recent paper by van den Berg et al. [2021] introduced a generalization of scoped effects that they call *latent effects* which supports a broader class of effects, including λ abstraction. While the

[22] A more extensive list of applications and frameworks can be found in Jeremy Yallop's Effects Bibliography: https://github.com/yallop/effects-bibliography

framework appears powerful, it currently lacks a denotational model, and seems to require similar weaving glue code as scoped effects. The solution we present in this paper does not require weaving glue code, and is given by a modular but simple mapping onto algebraic effects and handlers.

Looking beyond purely functional models of semantics and effects, there are also lines of work on modular support for side effects in operational semantics [Plotkin 2004]. Mosses' Modular Structural Operational Semantics [Mosses 2004] (MSOS) defines small-step rules that implicitly propagate an open-ended set of *auxiliary entities* which encode common classes of effects, such as reading or emitting data, stateful mutation, and even control effects [Sculthorpe et al. 2015]. The K Framework [Rosu and Serbanuta 2010] takes a different approach but provides many of the same benefits. These frameworks do not encapsulate operational details but instead make it notationally convenient to program (or specify semantics) with side-effects.

7 CONCLUSION

We have presented a new solution to the modularity problem with modeling and programming with higher-order effects. Our solution allows programming against an interface of higher-order effects in a way that provides effect encapsulation, meaning we can modularly change the implementation of effects without changing programs written against the interface and without changing the definition of any interface implementations. Furthermore, the solution requires a minimal amount of glue code to compose language definitions.

We have shown that the framework supports algebraic reasoning on a par with algebraic effects and handlers, albeit with some administrative overhead. While we have made use of Agda and dependent types throughout this paper, the framework should be straightforward to port to less dependently-typed functional languages, such as Haskell, OCaml, or Scala. An interesting direction for future work is to explore whether the framework could provide a denotational model for handling higher-order effects in standalone languages with support for effect handlers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments which helped improve the exposition of the paper. Furthermore, we thank Nicolas Wu, Andrew Tolmach, Peter Mosses, and Jaro Reinders for feedback on earlier drafts. This research was partially funded by the NWO VENI Composable and Safe-by-Construction Programming Language Definitions project (VI.Veni.192.259).

REFERENCES

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2620)*, Andrew D. Gordon (Ed.). Springer, 23–38. https://doi.org/10.1007/3-540-36576-1_2

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Michael A. Arbib and Ernest G. Manes. 1975. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press.

Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., USA.

Casper Bach Poulsen and Jaro Reinders. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects – Artifact. https://doi.org/10.5281/zenodo.7315899.

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

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

Richard S. Bird and Ross Paterson. 1999. Generalised folds for nested datatypes. *Formal Aspects Comput.* 11, 2 (1999), 200–222. https://doi.org/10.1007/s001650050047

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

Edwin C. Brady. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin C. Brady. 2013b. Programming and reasoning with algebraic effects and dependent types, See [Morrisett and Uustalu 2013], 133–144. https://doi.org/10.1145/2500365.2500581

Giuseppe Castagna and Andrew D. Gordon (Eds.). 2017. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. https://doi.org/10.1145/3009837

Pietro Cenciarelli and Eugenio Moggi. 1993. A syntactic approach to modularity in denotational semantics.

Koen Claessen. 1999. A Poor Man's Concurrency Monad. *J. Funct. Program.* 9, 3 (1999), 313–323. https://doi.org/10.1017/s0956796899003342

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 207–218. https://doi.org/10.1145/2429069.2429094

Andrzej Filinski. 1999. Representing Layered Monads. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 175–188. https://doi.org/10.1145/292540.292557

Marcelo P. Fiore and Sam Staton. 2014. Substitution, jumps, and algebraic effects. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 41:1–41:10. https://doi.org/10.1145/2603088.2603163

Peter G. Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1862)*, Peter Clote and Helmut Schwichtenberg (Eds.). Springer, 317–331. https://doi.org/10.1007/3-540-44622-2_21

Mauro Jaskelioff. 2008. Monatron: An Extensible Monad Transformer Library. In *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5836)*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer, 233–248. https://doi.org/10.1007/978-3-642-24452-0_13

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text (Lecture Notes in Computer Science, Vol. 925)*, Johan Jeuring and Erik Meijer (Eds.). Springer, 97–136. https://doi.org/10.1007/3-540-59451-5_4

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, Sam Lindley, and Nicolas Oury. 2013. Handlers in action, See [Morrisett and Uustalu 2013], 145–158. https://doi.org/10.1145/2500365.2500590

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. https://doi.org/10.1145/2804302.2804319

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects, See [Castagna and Gordon 2017], 486–499. https://doi.org/10.1145/3009837.3009872

Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.* 19, 4 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do, See [Castagna and Gordon 2017], 500–514. https://doi.org/10.1145/3009837.3009897

Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in proof theory, Vol. 1. Bibliopolis.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 124–144. https://doi.org/10.1007/3540543961_7

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

Eugenio Moggi. 1989b. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Greg Morrisett and Tarmo Uustalu (Eds.). 2013. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM. https://doi.org/10.1145/2500365

Peter D. Mosses. 2004. Modular structural operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 199–208. https://doi.org/10.1145/53990.54010

Benjamin C. Pierce. 1991. *Basic category theory for computer scientists*. MIT Press.

Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 308)*, Bart Jacobs, Alexandra Silva, and Sam Staton (Eds.). Elsevier, 273–288. https://doi.org/10.1016/j.entcs.2014.10.015

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.). ACM, 809–818. https://doi.org/10.1145/3209108.3209166

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2303)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

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

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

David Schmidt. 1986. *Denotational Semantics*. Allyn and Bacon.

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*, Richard A. Eisenberg (Ed.). ACM, 98–113. https://doi.org/10.1145/3331545.3342595

Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. 2014. Heuristics Entwined with Handlers Combined: From Functional Specification to Logic Programming Implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, Olaf Chitil, Andy King, and Olivier Danvy (Eds.). ACM, 259–270. https://doi.org/10.1145/2643135.2643145

Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses. 2015. A Modular Structural Operational Semantics for Delimited Continuations. In *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015 (EPTCS, Vol. 212)*, Olivier Danvy and Ugo de'Liguoro (Eds.). 63–80. https://doi.org/10.4204/EPTCS.212.5

Guy L. Steele Jr. 1994. Building Interpreters by Composing Monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 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

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Hayo Thielecke. 1997. *Categorical Structure of Continuation Passing Style*. Ph. D. Dissertation. University of Edinburgh.

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13008)*, Hakjoo Oh (Ed.). Springer, 182–201. https://doi.org/10.1007/978-3-030-89051-3_11

Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 1–14. https://doi.org/10.1145/143165.143169

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 1–12. https://doi.org/10.1145/2633357.2633358

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. Structured Handling of Scoped Effects. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 462–491. https://doi.org/10.1007/978-3-030-99336-8_17

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318

Received 2022-07-07; accepted 2022-11-07