JFP, 51 pages, 2021.
doi:10.1017/xxxxx

Hefty Algebras: Modular Elaboration of Higher-Order Effects

CAS VAN DER REST
Delft University of Technology
(e-mail: c.r.vanderrest@tudelft.nl)

CASPER BACH POULSEN
Delft University of Technology
(e-mail: c.b.poulsen@tudelft.nl)

Abstract

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. We demonstrate how our solution scales to define a wide range of known higher-order effects from the literature, and develop modular higher-order effect theories and modular reasoning principles that build on and extend the state of the art in modular algebraic effect theories. All results are formalized in Agda.

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, 1989*b*) have long been the preferred solution to this research question. However, *algebraic effects and handlers* (Plotkin & 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.

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 an argument and returns the unit value. Using algebraic effects and handlers its type is:

$$\mathit{out} : \mathit{String} \to () \mathrel{!} \mathit{Output}$$

Here *Output* is the *effect* of the operation. In general $A \mathrel{!} \Delta$ 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 $A \mathrel{!} \Delta$ 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 $() \mathrel{!} \mathit{Output}$ 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 $A \mathrel{!} \Delta \;\Rightarrow\; B \mathrel{!} \Delta'$, where Δ is the row of effects before applying the handler and Δ' is the row after. For example, here is a specific type of an effect handler for *Output*:

$$\mathit{hOut} : A \mathrel{!} \mathit{Output}, \Delta \Rightarrow (A \times \mathit{String}) \mathrel{!} \Delta$$

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:

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

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 $\mathit{out}\ s$ is then y and the current output (s) plus the output of the rest of the program (s').

In general, a computation $m : A \mathrel{!} \Delta$ can only be run in a context that provides handlers for each effect in Δ. To this end, the expression **with** h **handle** m represents applying the

[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 & McKinna (2019), the *effect lifting* of Biernacki *et al.* (2018), and the *effect tunneling* of Zhang & Myers (2019).

handler h to handle a subset of effects of m. For example, consider:

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

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

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

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*). As a simple example of a higher-order operation, say we want to define an effect *Censor* with a single operation *censor* with the following type, where A and Δ are implicitly universally quantified by the type signature:

$$\mathit{censor} : (\mathit{String} \to \mathit{String}) \to A \,!\, \mathit{Censor}, \Delta \to A \,!\, \mathit{Censor}, \Delta$$

The intended semantics for the operation $\mathit{censor}\ f\ m$ is to apply a censoring function $f : \mathit{String} \to \mathit{String}$ to the output printed by the computation m. In this section we explain how and why declaring and handling operations such as this using algebraic effects and handlers alone does not enjoy the same modularity benefits as the plain algebraic effects discussed in Section 1.1.

The lack of support for higher-order effects stems from how handler cases are typed. Following Plotkin & Pretnar (2009); Pretnar (2015), the left and right hand sides of handler cases are typed as follows:

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

Here, A is the argument type of an operation, and B is the return type of an operation. The term c represents the code of the handler case, which must have type $C!\Delta'$, for some overall handler return type C, and some remaining set of effects Δ'. The only way for c to have this type is if (1) $c = \textbf{return}\ w$, for some $w : C$; (2) if c calls the continuation k; or (3) if the operation argument type v has type $A = () \to C \,!\, \Delta'$. Here, option (3) seems most promising for encoding higher-order effects.

However, encoding computations as value arguments of operations in this way is non-modular. Following Plotkin & Pretnar (2009); Pretnar (2015), if h handles operations other than *op*, then

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

Consequently, if v contains effects of the type that h handles, then the handler of the operation $op\ v$ must eventually explicitly re-apply h or a different handler to handle those effects that h was supposed to handle. If we apply more handlers of effects contained in the value v, then the handler of $op\ v$ must eventually explicitly apply handlers for those too. This sensitivity to the order of applying handlers makes handling higher-order operations encoded in this way non-modular.

Another consequence of Eq. $(*)$ 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 & 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* (also known as *generic effects* (Plotkin & Power, 2003)); 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 $f : String \to String$.[2] Observe:

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

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

[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`

hOut:

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

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 & 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* (Wu *et al.*, 2014; Piróg *et al.*, 2018; 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 & 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 Section 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:

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

Here $A \,!!\, H$ 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 Section 3). Using $A \,!!\, H \Rrightarrow A \,!\, \Delta$ 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 $e_1 : A \,!!\, H_1 \Rrightarrow A \,!\, \Delta$ and $e_2 : A \,!!\, H_2 \Rrightarrow A \,!\, \Delta$ into an algebra $e_{12} : A \,!!\, H_1, H_2 \Rrightarrow A \,!\, \Delta$.[3]

Elaboration algebras are as simple to define as non-modular elaborations such as *censor* (Section 1.2). For example, here is the elaboration algebra for the higher-order *Censor* effect whose only associated operation is the higher-order operation $censor_{op} : (String \to$

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

$String) \to A \mathbin{!!} H \to A \mathbin{!!} H$:

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

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]

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

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*:

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

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":

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

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, 2013*a*); but also in less dependently typed languages like Haskell, OCaml, or Scala.[5] By working in 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:

- Section 2 describes how to encode algebraic effects in Agda, revisits the modularity problem with higher-order operations, and summarizes how scoped effects and

[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 (van der Rest & Poulsen, 2024) contains a shallow embedding of elaboration algebras in Haskell.

handlers address the modularity problem, for some (*scoped* operations) but not all higher-order operations.

- Section 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 $A\ !!\ H$ discussed in Section 1.3 correspond to hefty trees, and the elaborations of type $A\ !!\ H \Rrightarrow A\ !\ \Delta$ correspond to hefty algebras.
- Section 4 presents examples of how to define hefty algebras for common higher-order effects from the literature on effect handlers.
- Section 5 shows that hefty algebras support formal and modular reasoning on a par with algebraic effects and handlers, by developing reasoning infrastructure that supports verification of equational laws for higher-order effects such as exception catching. Crucially, proofs of correctness of elaborations are compositional. When composing two proven correct elaboration, correctness of the combined elaboration follows immediately without requiring further proof work.

Section 6 discusses related work and Section 7 concludes. The paper assumes a passing familiarity with dependent types. We do not assume familiarity with Agda: we explain Agda-specific syntax and features when we use them.

An artifact containing the code of the paper and a Haskell embedding of the same ideas is available online (van der Rest & Poulsen, 2024). A subset of the contributions of this paper were previously published in a conference paper (Poulsen & van der Rest, 2023). While that version of the paper too discusses reasoning about higher-order effects, the correctness proofs were non-modular, in that they make assumptions about the order in which the algebraic effects implementing a higher-order effect are handled. When combining elaborations, these assumptions are often incompatible, meaning that correctness proofs for the individual elaborations do not transfer to the combined elaboration. As a result, one would have to re-prove correctness for every combination of elaborations. For this extended version, we developed reasoning infrastructure to support modular reasoning about higher-order effects in Section 5, and proved that correctness of elaborations is preserved under composition of elaborations.

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. Sections 2.1 to 2.4 defines algebraic effects and handlers; Section 2.5 revisits the problem of defining higher-order effects using algebraic effects and handlers; and Section 2.6 discusses how scoped effects (Wu *et al.*, 2014; Piróg *et al.*, 2018; Yang *et al.*, 2022) solves the problem for *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; Swierstra, 2008; Kiselyov & Ishii, 2015; Wu *et al.*, 2014; Kammar *et al.*, 2013) 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. Effect signatures are given by a (dependent) record type:[9] [10]

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

Here, Op is the set of operations, and Ret defines the *return type* for each operation in the set Op. The extension of an effect signature, ⟦_⟧, reflects its input of type Effect as a value of type Set → Set:[11]

```
⟦_⟧ : Effect → Set → Set
⟦ Δ ⟧ X = Σ (Op Δ) λ op → Ret Δ op → X
```

The extension of an effect Δ into Set → Set is indeed a functor, as witnessed by the following function:[12]

```
map-sig : (X → Y) → ⟦ Δ ⟧ X → ⟦ Δ ⟧ Y
map-sig f (op , k) = ( op , f ∘ k )
```

As discussed in the introduction, computations may use multiple different effects. Effect signatures are closed under co-products:[13] [14]

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

[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, where an endofunctor is a functor whose domain and codomain coincides; e.g., Set → 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] Here, $\Sigma : (A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{Set}$ is a *dependen sum*.

[12] To show that this is truly a functor, we should also prove that map-sig satisfies the *functor laws*. We will not make use of these functor laws in this paper, so we omit them.

[13] 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.

[14] _⊎_ 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 $[\_,\_] : (A \to X) \to (B \to X) \to (A \uplus B) \to X$.

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

The syntax of computations with effects Δ is given by the free monad over Δ. We encode the free monad as follows:

```
data Free (Δ : Effect) (A : Set) : Set where
  pure   : A               → Free Δ A
  impure : ⟦ Δ ⟧ (Free Δ A) → Free Δ A
```

Here, pure is a computation with no side-effects, whereas impure is an operation whose syntax is given by the functor ⟦ Δ ⟧. By applying this functor to Free Δ A, we encode an operation whose *continuation* may contain more effectful operations.[15] To see in what sense, let us consider an example.

Example. 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 Section 1:

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

Section 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:[16]

```
hello-throw : Free (Output ⊕ Throw) A
hello-throw = impure (inj₁ (out "Hello") , λ _ →
```

[15] By unfolding the definition of ⟦_⟧ one can see that our definition of the free monad is identical to the I/O trees of Hancock & Setzer (2000), or the so-called *freer monad* of Kiselyov & Ishii (2015).

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

```
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 *smart constructor* constructs an effectful computation comprising a single operation. The type of this computation is polymorphic in what other effects the computation has. For example, the type of a smart constructor for the out effect is:

```
‵out : ⦃ Output ≲ Δ ⦄ → String → Free Δ ⊤
```

Here, the ⦃ Output ≲ Δ ⦄ type 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.[17] Thus, calling ‵out will automatically inject the Output effect into some larger effect row Δ.

We define the $\lesssim$ order on effect rows in terms of a different $\Delta_1 \bullet \Delta_2 \approx \Delta$ which witnesses that any operation of Δ is isomorphic to *either* an operation of Δ_1 *or* an operation of Δ_2:[18][19]

```
record _•_≈_ (Δ₁ Δ₂ Δ : Effect) : Set₁ where
  field reorder : ∀ {X} → ⟦ Δ₁ ⊕ Δ₂ ⟧ X ↔ ⟦ Δ ⟧ X
```

Using this, the $\lesssim$ order is defined as follows:

```
_≲_ : (Δ₁ Δ₂ : Effect) → Set₁
Δ₁ ≲ Δ₂ = Σ Effect (λ Δ′ → Δ₁ • Δ′ ≈ Δ₂)
```

It is straightforward to show that $\lesssim$ is a *preorder*; i.e., that it is a *reflexive* and *transitive* relation.

We can also define the following function, which uses a $\Delta_1 \lesssim \Delta_2$ witness to coerce an operation of effect type Δ_1 into an operation of some larger effect type Δ_2.[20]

```
inj : ⦃ Δ₁ ≲ Δ₂ ⦄ → ⟦ Δ₁ ⟧ A → ⟦ Δ₂ ⟧ A
inj ⦃ _ , w ⦄ (c , k) = w .reorder .to (inj₁ c , k)
```

Furthermore, we can freely coerce the operations of a computation from one effect row type to a different effect row type:[21] [22]

[17] 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`

[18] Here ∀ {X} is implicit universal quantification over an X : Set: `https://agda.readthedocs.io/en/v2.6.2.2/language/implicit-arguments.html`

[19] ↔ is the type of an *isomorphism* on Set from the Agda Standard Library. It is given by a record with two fields: the to field represents the → direction of the isomorphism, and from field represents the ← direction of the isomorphism.

[20] The dot notation w .reorder projects the reorder field of the record w.

[21] The notation ∀[_] is from the Agda Standard library, and is defined as follows: ∀[P] = ∀ x → P x.

[22] We can think of the hmap-free function as a "higher-order" map for Free: given a natural transformation between (the extension of) signatures, we can can transform the signature of a computation. This amounts

$$\begin{aligned}
&\mathsf{hmap\text{-}free} : \forall[\,⟦\,\Delta_1\,⟧ \Rightarrow ⟦\,\Delta_2\,⟧\,] \to \forall[\,\mathsf{Free}\ \Delta_1 \Rightarrow \mathsf{Free}\ \Delta_2\,]\\
&\mathsf{hmap\text{-}free}\ \theta\ (\mathsf{pure}\ x) = \mathsf{pure}\ x\\
&\mathsf{hmap\text{-}free}\ \theta\ (\mathsf{impure}\ (c\ ,\ k)) = \mathsf{impure}\ (\theta\ (c\ ,\ \mathsf{hmap\text{-}free}\ \theta \circ k))
\end{aligned}$$

Using this infrastructure, we can now implement a generic inject function which lets us define smart constructors for operations such as the out operation discussed in the previous subsection.

$$\begin{aligned}
&\mathsf{inject} : \{\!\!\{\ \Delta_1 \lesssim \Delta_2\ \}\!\!\} \to \mathsf{Free}\ \Delta_1\ A \to \mathsf{Free}\ \Delta_2\ A\\
&\mathsf{inject} = \mathsf{hmap\text{-}free}\ \mathsf{inj}
\end{aligned}$$

$$\begin{aligned}
&\text{`}\mathsf{out} : \{\!\!\{\ \mathsf{Output} \lesssim \Delta\ \}\!\!\} \to \mathsf{String} \to \mathsf{Free}\ \Delta\ \top\\
&\text{`}\mathsf{out}\ s = \mathsf{inject}\ (\mathsf{impure}\ (\mathsf{out}\ s\ ,\ \mathsf{pure}))
\end{aligned}$$

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.

$$\begin{aligned}
&\mathsf{fold} : (A \to B) \to \mathsf{Alg}\ \Delta\ B \to \mathsf{Free}\ \Delta\ A \to B\\
&\mathsf{fold}\ g\ a\ (\mathsf{pure}\ x) = g\ x\\
&\mathsf{fold}\ g\ a\ (\mathsf{impure}\ (op\ ,\ k)) = a\ (op\ ,\ \mathsf{fold}\ g\ a \circ k)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Alg} : (\Delta : \mathsf{Effect})\ (A : \mathsf{Set}) \to \mathsf{Set}\\
&\mathsf{Alg}\ \Delta\ A = ⟦\,\Delta\,⟧\ A \to A
\end{aligned}$$

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 & 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:

$$\begin{aligned}
&_\!\gg\!=\!_ : \mathsf{Free}\ \Delta\ A \to (A \to \mathsf{Free}\ \Delta\ B) \to \mathsf{Free}\ \Delta\ B\\
&m \gg= g = \mathsf{fold}\ g\ \mathsf{impure}\ m
\end{aligned}$$

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

Example. The following defines a smart constructor for throw:

$$\text{`}\mathsf{throw} : \{\!\!\{\ \mathsf{Throw} \lesssim \Delta\ \}\!\!\} \to \mathsf{Free}\ \Delta\ A$$

Using this and the definition of $\gg=$ above, we can use **do**-notation in Agda to make the hello-throw program from Section 2.1 more readable:

$$\begin{aligned}
&\mathsf{hello\text{-}throw}_1 : \{\!\!\{\ \mathsf{Output} \lesssim \Delta\ \}\!\!\} \to \{\!\!\{\ \mathsf{Throw} \lesssim \Delta\ \}\!\!\} \to \mathsf{Free}\ \Delta\ A\\
&\mathsf{hello\text{-}throw}_1 = \mathbf{do}\ \text{`}\mathsf{out}\ \texttt{"Hello"};\ \text{`}\mathsf{out}\ \texttt{" world!"};\ \text{`}\mathsf{throw}
\end{aligned}$$

to the observation that Free is a functor over the category of containers and container morphisms; assuming hmap-free preserves naturality.

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, interpreting 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* (Leijen, 2017, §2.2) defines how to fold respectively pure and impure computations:[23]

```
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 $P \rightarrow$ Free Δ' B. The following function does so by folding using ret, hdl, and a to-front function:[24]

```
to-front : ⦃ Δ₁ • Δ₂ ≈ Δ ⦄ → Free Δ A → Free (Δ₁ ⊕ Δ₂) A
to-front ⦃ w ⦄ = hmap-free (w .reorder .from)

given_handle_ : ⦃ w : Δ₁ • Δ₂ ≈ Δ ⦄
               → ⟨ A ! Δ₁ ⇒ P ⇒ B ! Δ₂ ⟩ → Free Δ A → (P → Free Δ₂ B)
given_handle_ h m = fold
  (ret h)
  ( λ where (inj₁ c , k) p → hdl h (c , k) p
            (inj₂ c , k) p → impure (c , flip k 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 $A \,!\, \Delta, \Delta' \Rightarrow P \rightarrow B \,!\, \Delta'$, 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 Section 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

[23] A simpler type of handler could omit the parameter; i.e., $\langle A \,!\, \Delta \Rightarrow B \,!\, \Delta' \rangle$, for some A,B : Set and Δ,Δ' : Effect. As demonstrated in, e.g., the work of Pretnar (2015, §2.4), this type of handler can leverage host language binding to handle, e.g., the *state effect* which we discuss later. The style of parameterized handler we use here follows the exposition of, e.g., Wu *et al.* (2014); Wu & Schrijvers (2015).

[24] The syntax λ **where** ... is a *pattern-matching* lambda in Agda. The function flip has the following type: $(A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)$.

```
data StateOp : Set where          State : Effect
  get :     StateOp               Op State = StateOp
  put : ℕ → 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 : ⦃ 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 (hSt below), and a simple test (incr-test, also below) which uses (the elided) smart constructors for get and put

difference, the handler is the same as in Section 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            un : Free Nil A → A
Op  Nil = ⊥             un (pure x) = x
Ret Nil = ⊥-elim
```

Using these, we can run a simple hello world program:[25]

```
hello′ : ⦃ Output ≲ Δ ⦄ → Free Δ ⊤     test-hello : un (given hOut handle hello′ $ tt)
hello′ = do                                          ≡ (tt , "Hello world!")
  `out "Hello"; `out " world!"         test-hello = refl
```

An example 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

Section 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. To this end, we use a different effect whose interface is summarized by the CatchM record below. The record asserts that the computation type M : Set → Set has at least a higher-order operation catch and a first-order operation throw:

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

$$
\begin{array}{l}
\textbf{record}\ \mathsf{CatchM}\ (M : \mathsf{Set} \to \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field}\ \mathsf{catch} : M\,A \to M\,A \to M\,A \\
\qquad\quad \mathsf{throw} : M\,A
\end{array}
$$

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 Section 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 Section 1.2, we can encode catch in terms of more primitive effects and handlers, such as the following handler for the Throw effect:

$$
\begin{array}{l}
\mathsf{hThrow} : \langle\, A \mathbin{!} \mathsf{Throw} \Rightarrow \top \Rightarrow (\mathsf{Maybe}\ A) \mathbin{!} \Delta' \,\rangle \\
\mathsf{ret}\ \ \mathsf{hThrow}\ x\ \_ = \mathsf{pure}\ (\mathsf{just}\ x) \\
\mathsf{hdl}\ \mathsf{hThrow}\ (\mathsf{throw}\ ,\ k)\ \_ = \mathsf{pure}\ \mathsf{nothing}
\end{array}
$$

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 constructor. 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:

$$
\sharp\_ : \{\!\{\ \Delta_1 \lesssim \Delta_2\ \}\!\} \to \mathsf{Free}\ \Delta_1\ A \to \mathsf{Free}\ \Delta_2\ A
$$

Using this, the following elaboration defines a semantics for the catch operation:[26] [27]

$$
\begin{array}{l}
\mathsf{catch} : \{\!\{\ \mathsf{Throw} \lesssim \Delta\ \}\!\} \to \mathsf{Free}\ \Delta\ A \to \mathsf{Free}\ \Delta\ A \to \mathsf{Free}\ \Delta\ A \\
\mathsf{catch}\ m_1\ m_2 = (\sharp\ (\mathsf{given}\ \mathsf{hThrow}\ \mathsf{handle}\ m_1)\ \mathsf{tt}) \gg\!= \mathsf{maybe}\ \mathsf{pure}\ m_2
\end{array}
$$

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 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.

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

[27] The instance resolution machinery of Agda requires some help to resolve the instance argument of $\sharp$ here. We provide a hint to Agda's instance resolution machinery in an implicit instance argument that we omit for readability in the paper. In the rest of this paper, we will occasionally follow the same convention.

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 Section 3.

Scoped effects extend the expressiveness of algebraic effects to support a class of higher-order operations that Wu *et al.* (2014); Piróg *et al.* (2018); 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) : Set where
  return : A                          → Prog Δ γ A
  call   : ⟦ Δ ⟧ (Prog Δ γ A)         → Prog Δ γ A
  enter  : ⟦ γ ⟧ (Prog Δ γ (Prog Δ γ A)) → Prog Δ γ A
```

Here, the enter constructor represents a higher-order operation with *sub-scopes*; i.e., computations that themselves return computations:

$$\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{outer}}\ (\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{inner}}\ A)$$

This type represents *scoped* computations in the sense that outer computations can be handled independently of inner ones, as we illustrate in Section 2.6.2. One way to think of inner computations is as continuations (or join-points) of sub-scopes.

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 → Set) → Set₁
CallAlg Δ G =
  {A : Set} → ⟦ Δ ⟧ (G A) → G A

EnterAlg : (γ : Effect) (G : Set → Set) → Set₁
EnterAlg γ G =
  {A B : Set} → ⟦ γ ⟧ (G (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 Section 2.4. The crucial addition which adds support for higher-order operations is the henter case.

The henter field is given by an EnterAlg case. This case takes as input a scoped operation whose outer and inner computation have been folded into a parameterized computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ X)$; and returns as output an interpretation of that operation as a computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ X)$. 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 : Δ₁ • Δ₂ ≈ Δ ⦄ ⦃ w₂ : γ₁ • γ₂ ≈ γ ⦄
                       → ⟨•! Δ₁ ! γ₁ ⇒ P ⇒ G •! Δ₂ ! γ₂ ⟩
                       → Prog Δ γ A → P → Prog Δ₂ γ₂ (G A)
given h handle-scoped m = hcata (ret h)
  ⊕[ hcall h
   , (λ where (c , k) p → call (c , flip k p)) ]
  ⊕[ (λ {A} → henter h {A})
   , (λ where (c , k) p → enter (c , λ x → map-prog (λ y → glue h id y p) (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 $\to$ Set of a parameterized scoped effect handler.

The scoped effect handler for exception catching is thus:

```
hCatch : ⟨•! Throw ! Catch ⇒ ⊤ ⇒ Maybe •! Δ ! γ ⟩
ret     hCatch x _ = return (just x)
hcall   hCatch (throw , k) _ = return nothing
henter  hCatch (catch , k) _ = let m₁ = k true
                                   m₂ = k false in
  m₁ tt ≫= λ where
    (just f)  → f tt
    nothing → m₂ tt ≫= maybe (_$ tt) (return nothing)
glue hCatch k x _ = maybe (flip k tt) (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.

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 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 enter constructor of Prog:

$$\text{enter} : [\![\, \gamma \,]\!] \; (\underbrace{\text{Prog}\,\Delta\,\gamma}_{\text{outer}} \; (\underbrace{\text{Prog}\,\Delta\,\gamma}_{\text{inner}}\; A)) \to \text{Prog}\,\Delta\,\gamma\,A$$

As summarized earlier in this subsection, enter lets us represent higher-order operations (specifically, *scoped operations*), whereas call does not (only *algebraic operations*). Just like we defined the computational parameters as scopes (given by the outer Prog in the type of enter), we might try to define the body of a lambda as a scope in a similar way. However, whereas the catch operation always passes control to its continuation (the inner Prog), the lam effect is supposed to package the body of the lambda into a value and pass this value to the continuation (the inner computation). Because the inner computation is nested within the outer computation, *the only way to gain access to the inner computation (the continuation) is by first running the outer computation (the body of the lambda)*. This does not give us the right semantics.

It is possible to elaborate the LambdaM operations into more primitive effects and handlers, but as discussed in Sections 1.2 and 2.5, such elaborations are not modular. In the next section we show how to make such elaborations modular.

3 Hefty Trees and Algebras

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

To this end, we first introduce a new type of abstract syntax trees (Sections 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*; Section 3.4) over these trees. The following pipeline summarizes the idea, where H is a *higher-order effect signature*:

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

For the categorically inclined reader, Hefty conceptually corresponds to the initial algebra of the functor $HeftyF\ H\ A\ R = A + H\ R\ (R\ A)$ 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 $map : (X \to Y) \to H\ F\ X \to H\ F\ Y$ for any functor F as well as a function $hmap : \mathrm{Nat}(F, G) \to \mathrm{Nat}(H\ F, H\ G)$ 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 Section 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 a container-inspired approach to represent higher-order signature functors H as a strictly positive type. We discuss and compare our approach with previous work in Section 3.5.

3.1 Generalizing **Free** *to Support Higher-Order Operations*

As summarized in Section 2.1, Free $\Delta\ A$ 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 (Section 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. But the computation parameter of censor_{op} is *not* a continuation, since

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

The crux of the issue is how signature functors $[\![\,\Delta\,]\!] : \mathsf{Set} \to \mathsf{Set}$ are defined. Since this is an endofunctor on Set, the only suitable option in the impure constructor is to apply the functor to the type of *continuations*:

$$\mathsf{impure} : [\![\,\Delta\,]\!]\,(\underbrace{\mathsf{Free}\,\Delta\,A}_{\text{continuation}}) \to \mathsf{Free}\,\Delta\,A$$

A more flexible approach would be to allow signature functors to build computation trees with an *arbitrary return type*, including the return type of the continuation. A *higher-order* signature functor of some higher-order signature H, written $[\![\,H\,]\!]^{\mathsf{H}} : (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set} \to \mathsf{Set}$, would fit that bill. Using such a signature functor, we could define the impure case as follows:

$$\mathsf{impure} : [\![\,H\,]\!]^{\mathsf{H}}\;(\;\underbrace{\mathsf{Hefty}\,H}_{\substack{\text{computation}\\\text{type}}}\;)\;\;\overbrace{A}^{\substack{\text{continuation}\\\text{return type}}}\;\; \to \mathsf{Hefty}\,H\,A$$

Here, Hefty is the type of the free monad using higher-order signature functors instead. In the rest of this subsection, we consider how to define higher-order signature functors H, their higher-order functor extensions $[\![\_]\!]^{\mathsf{H}}$, and the type of Hefty trees.

Recall how we defined plain algebraic effects in terms of *containers*:

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

Here, Op is the type of operations, and Ret defines the return type of each operation. In order to allow operations to have sub-computations, we generalize this type to allow each operation to be associated with a number of sub-computations, where each sub-computation can have a different return type. The following record provides this generalization:

```
record Effectᴴ : Set₁ where
  field Opᴴ  : Set                               -- As before
        Retᴴ : Opᴴ → Set                         -- As before
        Fork : Opᴴ → Set                         -- New
        Ty   : {op : Opᴴ} (ψ : Fork op) → Set    -- New
```

The set of operations is still given by a type field (Op^{H}), and each operation still has a return type ($\mathsf{Ret}^{\mathsf{H}}$). Fork associates each operation with a type that indicates how many sub-computations (or *forks*) an operation has, and Ty indicates the return type of each such fork. For example, say we want to encode an operation *op* with two sub-computations with different return types, and whose return type is of a unit type. That is, using M as our type of computations:

$$op : M\,\mathbb{Z} \to M\,\mathbb{N} \to M\,\top$$

The following signature declares a higher-order effect signature with a single operation with return type ⊤, and with two forks (we use Bool to encode this fact), and where each fork has, respectively ℤ and ℕ as return types.

```
example-op : Effectᴴ
Opᴴ  example-op       = ⊤    -- A single operation
Retᴴ example-op tt    = ⊤    -- with return type ⊤
Fork example-op tt    = Bool -- with two forks
Ty   example-op false = ℤ    -- one fork has return type ℤ
Ty   example-op true  = ℕ    -- the other has return type ℕ
```

The extension of higher-order effect signatures implements the intuition explained above:

```
⟦_⟧ᴴ : Effectᴴ → (Set → Set) → Set → Set
⟦ H ⟧ᴴ M X =
  Σ (Opᴴ H) λ op → (Retᴴ H op → M X) × ((ψ : Fork H op) → M (Ty H ψ))
```

Let us unpack this definition.

$$\underbrace{\Sigma\,(\,\mathsf{Op}^{\mathsf{H}}\,H)\,\lambda\,op \rightarrow}_{(1)}\,(\underbrace{\mathsf{Ret}^{\mathsf{H}}\,H\,op \rightarrow M\,X}_{(2)})\times(\underbrace{(\psi : \mathsf{Fork}\,H\,op)}_{(3)} \rightarrow \underbrace{M\,(\mathsf{Ty}\,H\,\psi)}_{(4)})$$

The extension of a higher-order signature functor is given by (1) the sum of operations of the signature, where each operation has (2) a continuation (of type $M\ X$) that expects to be passed a value of the operation's return type, and (3) a set of forks where each fork is (4) a computation that returns the expected type for each fork.

Using the higher-order signature functor and its extension above, our generalized free monad becomes:

```
data Hefty (H : Effectᴴ) (A : Set) : Set where
  pure   : A → Hefty H A
  impure : ⟦ H ⟧ᴴ (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 Section 2.2, Fig. 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 & Paterson, 1999). The fold of a nested data type is limited to describe *natural transformations*. As Bird & 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 , k , s) ≫= g = impure (op , (_≫= g) ∘ k , s)
```

```
data CensorOp : Set where                 Censor : Effectᴴ
  censor :  (String → String)             Opᴴ  Censor                = CensorOp
            → CensorOp                     Retᴴ Censor (censor f)     = ⊤
                                           Fork Censor (censor f)     = ⊤
                                           Ty   Censor {censor f} tt  = ⊤
```

```
censor_op : (String → String) → Hefty (Censor ∔ H) ⊤ → Hefty (Censor ∔ H) ⊤
censor_op f m = impure (inj₁ (censor f) , (λ where tt → 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)

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

In Section 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 Section 3.2) how Hefty trees support programming against an interface of both algebraic and higher-order operations. We also address (in Section 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 → Effectᴴ
Opᴴ  (Lift Δ) = Op Δ
Retᴴ (Lift Δ) = Ret Δ
Fork (Lift Δ) = λ _ → ⊥
Ty   (Lift Δ) = λ()
```

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

```
↑_ : ⦃ w : Lift Δ ≲ᴴ 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:[28]

```
data CatchOpᵈ : Set₁ where
  catchᵈ : Set → CatchOpᵈ
```

```
Catchᵈ : Effectᴴ
Opᴴ  Catchᵈ = CatchOpᵈ
Retᴴ Catchᵈ (catchᵈ A) = A
Fork Catchᵈ (catchᵈ A) = Bool
Ty   Catchᵈ {catchᵈ 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 $\mathsf{CatchOp}^d$ has a constructor that quantifies over a type (Set), the $\mathsf{CatchOp}^d$ type lives in Set_1. Consequently it does not fit the definition of $\mathsf{Effect}^{\mathsf{H}}$, whose operations live in Set. There are two potential solutions to this problem: (1) increase the universe level of $\mathsf{Effect}^{\mathsf{H}}$ to allow Op^{H} to live in Set_1; or (2) use a *universe of types* (Martin-Löf, 1984). Either solution is applicable here; we choose type universes.

A universe of types is a (dependent) pair of a syntax of types (Ty : Set) and a semantic function ($[\![\_]\!]^{\mathsf{T}}$: Ty → Set) defining the meaning of the syntax by reflecting it into Agda's Set:

```
record Univ : Set₁ where
  field Type : Set
        ⟦_⟧ᵀ  : Type → Set
```

Section 4.1 contains a concrete example usage this notion of type universe. 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* ($[\![\ t\ ]\!]^{\mathsf{T}}$) as the return type of the computation parameters in the effect signature on the right below:

```
data CatchOp ⦃ u : Univ ⦄ : Set where
  catch : Type → CatchOp
```

```
Catch : ⦃ u : Univ ⦄ → Effectᴴ
Opᴴ  Catch           = CatchOp
Retᴴ Catch (catch t) = ⟦ t ⟧ᵀ
Fork Catch (catch t) = Bool
Ty   Catch {catch t} = λ _ → ⟦ 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.

[28] *d* is for *dubious*.

3.4 Hefty Algebras

As shown in Section 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 ($\mathsf{Alg}^{\mathsf{H}}$) and fold (or *catamorphism* (Meijer *et al.*, 1991), $\mathsf{cata}^{\mathsf{H}}$) for Hefty:

```
record Algᴴ (H : Effectᴴ) (F : Set → Set) : Set₁ where
  field alg : ⟦ H ⟧ᴴ F A → F A

cataᴴ : (∀ {A} → A → F A) → Algᴴ H F → Hefty H A → F A
cataᴴ g a (pure x)              = g x
cataᴴ g a (impure (op , k , s)) = alg a (op , ((cataᴴ g a ∘ k) , (cataᴴ g a ∘ s)))
```

Here $\mathsf{Alg}^{\mathsf{H}}$ 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:

```
_⋎_ : Algᴴ H₁ F → Algᴴ H₂ F → Algᴴ (H₁ ∔ H₂) F
alg (A₁ ⋎ A₂) (inj₁ op , k , s) = alg A₁ (op , k , s)
alg (A₁ ⋎ A₂) (inj₂ op , k , s) = alg A₂ (op , k , s)
```

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

```
Elaboration : Effectᴴ → Effect → Set₁
Elaboration H Δ = Algᴴ H (Free Δ)
```

An Elaboration H Δ 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 = cataᴴ pure
```

Example. The elaboration below is analogous to the non-modular catch elaboration discussed in Section 2.5 and in the beginning of this subsection:

```
eCatch : ⦃ u : Univ ⦄ ⦃ w : Throw ≲ Δ ⦄ → Elaboration Catch Δ

module _ ⦃ u : Univ ⦄ ⦃ w : Throw ≲ Δ ⦄ where
  eCatch : Elaboration Catch Δ

alg eCatch (catch t , k , s) =
  (♯ ((given hThrow handle s true) tt)) ≫= maybe k (s false ≫= k)
  where instance _ = _ , •-comm (w .proj₂)
```

The elaboration is essentially the same as its non-modular counterpart, except that it now uses the universe of types encoding discussed in Section 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 Fig. 1, the throw effect from Section 2.1, and the catch effect defined here:

```
transact : ⦃ wₛ : Lift State ≲ᴴ H ⦄ ⦃ wₜ : Lift Throw ≲ᴴ H ⦄ ⦃ w : Catch ≲ᴴ 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 : ⦃ _ : Throw ≲ Δ ⦄ ⦃ _ : State ≲ Δ ⦄
          → Elaboration (Catch ∔ Lift Throw ∔ Lift State ∔ Lift Nil) Δ
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 Section 4.2.2 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. As we show in Section 5, hefty algebras also let us define higher-order effect theories, akin to algebraic effect theories.

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 (Section 4.2.2) by Thielecke (1997); Fiore & Staton (2014), 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, $\mathsf{Effect}^{\mathsf{H}}$, 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.

Recent work by van den Berg & Schrijvers (2023) introduced a higher-order free monad that coincides with our Hefty type. Their work shows that hefty trees are, in fact, a free monad. Furthermore, they demonstrate that a range of existing effects frameworks from the literature can be viewed as instances of hefty trees.

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 Section 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 Section 4.1 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.:

$$\mathsf{Prog}\ \Delta_0\ \gamma_0\ A_0 \xrightarrow{h'_1} \mathsf{Prog}\ \Delta_1\ \gamma_1\ A_1 \xrightarrow{h'_2} \cdots \xrightarrow{h'_n} \mathsf{Prog}\ \mathsf{Nil}\ \mathsf{Nil}\ A_n \qquad (\ddagger)$$

As discussed in Section 2.6.3, each handler application modularly "weaves" effects through sub-computations, using a dedicated glue function.applying different scoped effect handlers in different orders.

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.:

$$\mathsf{Hefty}\ (H_0 \mathbin{\dot{+}} \cdots \mathbin{\dot{+}} H_m)\ A \xrightarrow{\mathsf{elaborate}\ (E_0\ \curlyvee\ \cdots\ \curlyvee\ E_m)} \mathsf{Free}\ \Delta\ A \xrightarrow{h_1} \cdots \xrightarrow{h_k} \mathsf{Free}\ \mathsf{Nil}\ A_k \qquad (\S)$$

The algebraic effect handlers $h_1, \ldots, h_k$ in (‡) serve the same purpose as the scoped effect handlers $h'_1, \ldots, h'_n$ in (§); namely, to provide a semantics of operations. The order of handling is significant for both algebraic effect handlers and for scoped effect handlers: applying the same handlers in different orders may give a different semantics.

In contrast, the order that elaborations $(E_1, \ldots, E_m)$ are composed in (§) does not matter. Hefty algebras merely mediate higher-order operations into "first-order" effect trees that then must be handled, using standard effect handlers. While scoped effects supports "weaving", standard algebraic effect handlers do not. This might suggest that scoped effects and handlers are generally more expressive. However, many scoped effects and handlers can be emulated using algebraic effects and hanlders, by encoding scoped operations as algebraic operations whose continuations encode a kind of scoped syntax, inspired by Wu *et al.* (2014, §7-9).[29] We illustrate how in Section 4.2.2.

4 Examples

As discussed in Section 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 (van der Rest & Poulsen, 2024) contains the full examples.

4.1 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 LamUniv : Set₁ where
  field ⦃ u ⦄ : Univ
        _↣_  : Type → Type → Type
        c    : Type → Type
```

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, given by the higher-order effect signature in Fig. 3, and summarized by the following smart constructors:

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

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 ⟧ᵀ). 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 that of the function value parameter type.

[29] We suspect that it is generally possible to encode scoped syntax and handlers in terms of algebraic operations and handlers, but verifying this is future work.

```
data LamOp ⦃ l : LamUniv ⦄ : Set where
  lam : {t₁ t₂ : Type} → LamOp
  var  : {t : Type}     → ⟦ c t ⟧ᵀ           → LamOp
  app : {t₁ t₂ : Type} → ⟦ (c t₁) ↣ t₂ ⟧ᵀ → LamOp

Lam : ⦃ l : LamUniv ⦄ → Effectᴴ
Opᴴ  Lam                        = LamOp
Retᴴ Lam (lam {t₁} {t₂})        = ⟦ (c t₁) ↣ t₂ ⟧ᵀ
Retᴴ Lam (var {t} _)            = ⟦ t ⟧ᵀ
Retᴴ Lam (app {t₁} {t₂} _)      = ⟦ t₂ ⟧ᵀ
Fork Lam (lam {t₁} {t₂})        = ⟦ c t₁ ⟧ᵀ
Fork Lam (var _)                = ⊥
Fork Lam (app {t₁} {t₂} _)      = ⊤
Ty   Lam {lam {t₁} {t₂}} _      = ⟦ t₂ ⟧ᵀ
Ty   Lam {var _} ()
Ty   Lam {app {t₁} {t₂} _} _ = ⟦ t₁ ⟧ᵀ
```

Fig. 3. Higher-order effect signature of λ abstraction and application

The interface above defines a kind of *higher-order abstract syntax* (Pfenning & 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.

4.1.1 Call-by-Value

We give a call-by-value interpretation of `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):

```
iso₁ : {t₁ t₂ : Type} → ⟦ t₁ ↣ t₂ ⟧ᵀ ↔ (⟦ t₁ ⟧ᵀ → Free Δ ⟦ t₂ ⟧ᵀ)
iso₂ : {t : Type}     → ⟦ c t      ⟧ᵀ ↔ ⟦ 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 that of 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 , k , ψ) = k (from ψ)
```

```
alg eLamCBV (var x , k , _) = k (to x)
alg eLamCBV (app f , k , ψ) = 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 $[\![\,\mathsf{c}\ t_1\,]\!]^{\mathsf{T}} \to \mathsf{Free}\ \Delta\ [\![\,t_2\,]\!]^{\mathsf{T}}$ to a value type $[\![\,(\mathsf{c}\ t_1) \rightarrowtail t_2\,]\!]^{\mathsf{T}}$, using the isomorphism iso_1. The var case uses the to function to mediate a $[\![\,\mathsf{c}\ t\,]\!]^{\mathsf{T}}$ value to a $[\![\,t\,]\!]^{\mathsf{T}}$ 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 $[\![\,num\,]\!]^{\mathsf{T}} \leftrightarrow \mathbb{N}$:

```
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 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
```

4.1.2 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:

$\mathsf{iso}_1 : \{t_1\ t_2 : \mathsf{Type}\} \to [\![\ t_1 \rightarrowtail t_2\ ]\!]^{\mathsf{T}} \leftrightarrow ([\![\ t_1\ ]\!]^{\mathsf{T}} \to \mathsf{Free}\ \Delta\ [\![\ t_2\ ]\!]^{\mathsf{T}})$
$\mathsf{iso}_2 : \{t : \mathsf{Type}\} \quad\quad \to [\![\ \mathsf{c}\ t\ ]\!]^{\mathsf{T}} \quad\quad \leftrightarrow \mathsf{Free}\ \Delta\ [\![\ t\ ]\!]^{\mathsf{T}}$

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

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

4.2 Optionally Transactional Exception Catching

A feature of scoped effect handlers (Wu *et al.*, 2014; Piróg *et al.*, 2018; 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 Section 3.4 in connection with this transact program:

```
transact : ⦃ w_s : Lift State ≲ᴴ H ⦄ ⦃ w_t : Lift Throw ≲ᴴ H ⦄ ⦃ w : Catch ≲ᴴ 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 & 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,

$$\textbf{data}\ \mathsf{CCOp}\ \{\!\!\{\ u : \mathsf{Univ}\ \}\!\!\}\ (\mathit{Ref} : \mathsf{Type} \to \mathsf{Set}) : \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{sub}\ \ : \{t : \mathsf{Type}\} \to \mathsf{CCOp}\ \mathit{Ref}$$
$$\quad \mathsf{jump} : \{t : \mathsf{Type}\}\ (\mathit{ref} : \mathit{Ref}\ t)\ (x : [\![\ t\ ]\!]^{\mathsf{T}}) \to \mathsf{CCOp}\ \mathit{Ref}$$

$$\mathsf{CC} : \{\!\!\{\ u : \mathsf{Univ}\ \}\!\!\}\ (\mathit{Ref} : \mathsf{Type} \to \mathsf{Set}) \to \mathsf{Effect}$$
$$\mathsf{Op}\ (\mathsf{CC}\ \mathit{Ref}) = \mathsf{CCOp}\ \mathit{Ref}$$
$$\mathsf{Ret}\ (\mathsf{CC}\ \mathit{Ref})\ (\mathsf{sub}\ \{t\}) \quad = \mathit{Ref}\ t \uplus [\![\ t\ ]\!]^{\mathsf{T}}$$
$$\mathsf{Ret}\ (\mathsf{CC}\ \mathit{Ref})\ (\mathsf{jump}\ \mathit{ref}\ x) = \bot$$

Fig. 4. Effect signature of the sub/jump effect

the eCatch elaboration in Section 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 Thielecke (1997); Fiore & Staton (2014). 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.

4.2.1 Sub/Jump

We recall how to define two operations, sub and jump, due to Thielecke (1997); Fiore & Staton (2014). We define these operations as algebraic effects following Schrijvers *et al.* (2019). The algebraic effect signature of CC *Ref* is given in Fig. 4, and is summarized by the following smart constructors:

$$\text{`}\mathsf{sub}\ \ : \{\!\!\{\ w : \mathsf{CC}\ \mathit{Ref} \lesssim \Delta\ \}\!\!\}\ (b : \mathit{Ref}\ t \to \mathsf{Free}\ \Delta\ A)\ (k : [\![\ t\ ]\!]^{\mathsf{T}} \to \mathsf{Free}\ \Delta\ A) \to \mathsf{Free}\ \Delta\ A$$
$$\text{`}\mathsf{jump} : \{\!\!\{\ w : \mathsf{CC}\ \mathit{Ref} \lesssim \Delta\ \}\!\!\}\ (\mathit{ref} : \mathit{Ref}\ t)\ (x : [\![\ t\ ]\!]^{\mathsf{T}}) \to \mathsf{Free}\ \Delta\ B$$

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 $[\![\ t\ ]\!]^{\mathsf{T}}$. The `jump operation invokes such continuations.

The operations and their handler (abbreviated to h) satisfy the following laws:

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

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

```
data ChoiceOp : Set where
  or   : ChoiceOp
  fail : ChoiceOp
```

```
Choice : Effect
Op Choice = ChoiceOp
Ret Choice or = Bool
Ret Choice fail = ⊥
```

Fig. 5. Effect signature of the choice effect

4.2.2 *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 : CC Ref ≲ Δ ⦄ ⦃ w₂ : Throw ≲ Δ ⦄ → Elaboration Catch Δ
alg eCatchOT (catch x , k , ψ) = 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 handle the state effect before the sub/jump effect: 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 Section 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.

4.3 *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 effect signature for Choice is given in Fig. 5. The following smart constructors are the lifted higher-order counterparts to the ‘or and ‘fail operations:

```
_‘orᴴ_ : ⦃ Lift Choice ≲ᴴ H ⦄ → Hefty H A → Hefty H A → Hefty H A
‘failᴴ : ⦃ Lift Choice ≲ᴴ 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, whose higher-order effect signature is given in Fig. 6, is not an algebraic effect, but a scoped (and thus higher-order) effect.

```
data OnceOp ⦃ u : Univ ⦄ : Set where
  once : {t : Type} → OnceOp
```

```
Once : ⦃ u : Univ ⦄ → Effectᴴ
Opᴴ  Once = OnceOp
Retᴴ Once (once {t}) = ⟦ t ⟧ᵀ
Fork Once (once {t}) = ⊤
Ty   Once {once {t}} _ = ⟦ t ⟧ᵀ
```

Fig. 6. Higher-order effect signature of the once effect

```
‵once : ⦃ w : Once ≲ᴴ H ⦄ {t : Type} → Hefty H ⟦ t ⟧ᵀ → Hefty H ⟦ t ⟧ᵀ
```

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

```
eOnce : ⦃ Choice ≲ Δ ⦄ → Elaboration Once Δ
alg eOnce (once , k , ψ) = 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.

4.4 Concurrency

Finally, we consider how to define higher-order operations for concurrency, inspired by Yang *et al.*'s [2022] *resumption monad* (Claessen, 1999; Schmidt, 1986; Piróg & Gibbons, 2014) defined using scoped effects. We summarize our encoding and compare it with the resumption monad. The goal is to define the two operations, whose higher-order effect signature is given in **??**, and summarized by these smart constructors:

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

```
interleaveₗ : {Ref : Type → 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 Section 4.2.2. The interleave_l function ensures atomic execution by only interleaving code that is not

```
data ConcurOp ⦃ u : Univ ⦄ : Set where
  spawn : (t : Type) → ConcurOp
  atomic : (t : Type) → ConcurOp
```

```
Concur : ⦃ u : Univ ⦄ → Effectᴴ
Opᴴ Concur = ConcurOp
Retᴴ Concur (spawn t) = ⟦ t ⟧ᵀ
Retᴴ Concur (atomic t) = ⟦ t ⟧ᵀ

Fork Concur (spawn t) = Bool
Fork Concur (atomic t) = ⊤
Ty Concur {spawn t} _ = ⟦ t ⟧ᵀ
Ty Concur {atomic t} _ = ⟦ t ⟧ᵀ
```

Fig. 7. Higher-order effect signature of the concur effect

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 : ⦃ w : CC Ref ≲ Δ ⦄ → Elaboration Concur Δ
alg eConcur (spawn t , k , ψ) =
  from-front (interleaveₗ (to-front (ψ true)) (to-front (ψ false))) ≫= k
alg eConcur (atomic t , k , ψ) = ‘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 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. 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 $\mathsf{Alg}^{\mathsf{H}}$ 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.

5 Modular Reasoning for Higher-Order Effects

A key aspect of algebraic effects and handlers is the ability to state and prove *equational laws* that characterize correct implementations of effectful operations. Usually, an effect comes equipped with multiple laws that govern its intended behavior. An effect and its laws constitute an *effect theory* (Hyland *et al.*, 2006; Plotkin & Power, 2002, 2003; Yang & Wu, 2021). This concept of effect theory extends to *higher-order effect theories*, which describe the intended behavior of higher-order effects. In this section, we first discuss how to define theories for algebraic effects in Agda by adapting the exposition of Yang & Wu (2021), and show how correctness of implementations with respect to a given theory can be stated and proved. We then extend this reasoning infrastructure to higher-order effects, allowing for modular reasoning about the correctness of elaborations of higher-order effects.

Let us consider the state effect as an example, which comprises the get and put operations. With the state effect, we typically associate a set of equations (or laws) that specify how its implementations ought to behave. One such law is the *get-get* law, which captures the intuition that the state returned by two subsequent get operations does not change if we do not use the put operation in between:

$$`\mathsf{get} \gg\!= \lambda s \rightarrow `\mathsf{get} \gg\!= \lambda s' \rightarrow k\ s\ s' \ \equiv\ `\mathsf{get} \gg\!= \lambda s \rightarrow k\ s\ s$$

We can define equational laws for higher-order effects in a similar fashion. For example, the following *catch-return* law for the `catch operation of the Catch effect, stating that catching exceptions in a computation that only returns a value does nothing.

$$`\mathsf{catch}\ (\mathsf{pure}\ x)\ m\ \equiv\ \mathsf{pure}\ x$$

Correctness of an implementation of an algebraic effect with respect to a given theory is defined by comparing the implementations of programs that are equal under that theory. That is, if we can show that two programs are equal using the equations of a theory for its effects, handling the effects should produce equal results. For instance, a way to implement the state effect is by mapping programs to functions of the form $S \rightarrow S \times A$. Such an implementation would be correct if programs that are equal with respect to a theory of the state effect are mapped to functions that give the same value and output state for every input state.

For higher-order effects, correctness is defined in a similar manner. However, since we define higher-order effects by elaborating them into algebraic effects, correctness of elaborations with respect to a higher-order effect theory is defined by comparing the elaborated programs. Crucially, the elaborated programs do not have to be syntactically equal, but rather we should be able to prove them equal using a theory of the algebraic effects used to implement a higher-order effect.

Effect theories are known to be closed under the co-product of effects, by combining the equations into a new theory that contains all equations for both effects (Hyland *et al.*, 2006). Similarly, theories of higher-order effects are closed under sums of higher-order effect signatures. In Section 5.8, we show that composing two elaborations preserves their correctness, with respect to the sum of their respective theories.

5.1 Theories of Algebraic Effects

Theories of effects are collections of equations, so we start defining the type of equations in Agda. At its core, an equation for an effect Δ is given by a pair of effect trees of type Free Δ A, that define the left- and right-hand side of the equation. However, looking at the *get-get* law above, we see that this equation contains a *term metavariable*; i.e., k. Furthermore, when considering the type of k, which is $S \rightarrow S \rightarrow$ Free Δ A, we see that it refers to a *type metavariable*; i.e., A. Generally speaking, an equation may refer to any number of term metavariables, which, in turn, may depend on any number of type metavariables. Moreover, the type of the value returned by the left hand side and right hand side of an equation may depend on these type metavariables as well, as is the case for the *get-get* law above. This motivates the following definition of equations in Agda.

```
record Equation (Δ : Effect) : Set₁ where
  field
    V       : ℕ
    Γ       : Vec Set V → Set
    R       : Vec Set V → Set
    lhs rhs : (vs : Vec Set V) → Γ vs → Free Δ (R vs)
```

An equation consists of five components. The field V defines the number of type metavariables used in the equation. Then, the fields Γ and R respectively define the term metavariables (Vec Set V → Set) and return type (Vec Set V → Set) of the equation.

Example. To illustrate how the Equation record captures equational laws of effects, we consider how to define the *get-get* as a value of type Equation State.

```
get-get : Equation State
V   get-get = 1
Γ   get-get = λ where (A :: []) → ℕ → ℕ → Free State A
R   get-get = λ where (A :: []) → A
lhs get-get (A :: []) k = `get ≫= λ s → `get ≫= λ s′ → k s s′
rhs get-get (A :: []) k = `get ≫= λ s → k s s
```

The fields lhs and rhs define the left- and right-hand sides of the equation. Both sides only use a single term metavariable, representing a continuation of type $\mathbb{N} \to \mathbb{N} \to$ Free State A. The field Γ declares this term meta-variable. For equations with more than $n > 1$ metavariables, we would define Γ as an n-ary product instead.

5.2 Modal Necessity

The current definition of equations is too weak, in the sense that it does not apply in many situations where it should. The issue is that it fixes the set of effects that can be used in the left- and right-hand side. To illustrate why this is problematic, consider the following equality:

$$\text{get} \gg\!= \lambda s \to \text{get} \gg\!= \lambda s' \to \text{throw} \ \equiv\ \text{get} \gg\!= \lambda s \to \text{throw} \tag{5.1}$$

We might expect to be able to prove this equality using the *get-get* law, but using the embedding of the law defined above—i.e., get-get—this is not possible. The reason for this is that we cannot pick an appropriate instantiation for the term metavariable k: it ranges over values of type $S \to S \to$ Free State A, inhibiting all references to effectful operation that are not part of the state effect, such as throw.

Given an equation for the effect Δ, the solution to this problem is to view Δ as a *lower bound* on the effects that might occur in the left-hand and right-hand side of the equation, rather than an exact specification. Effectively, this means that we close over all posible contexts of effects in which the equation can occur. This pattern of closing over all possible extensions of a type index is well-known (Allais *et al.*, 2021; van der Rest *et al.*, 2022), and corresponds to a shallow embedding of the Kripke semantics of the necessity modality from modal logic. We can define it in Agda as follows.[30]

```
record □ (P : Effect → Set₁) (Δ : Effect) : Set₁ where
  constructor necessary
  field
    □⟨_⟩ : ∀ {Δ′} → ⦃ Δ ≲ Δ′ ⦄ → P Δ′
```

Intuitively, the □ modality transforms, for any effect-indexed type (P : Effect → Set_1), an *exact* specification of the set of effects to a *lower bound* on the set of effects. For equations, the difference between terms of type Equation Δ and □ Equation Δ amounts to the former defining an equation relating programs that have exactly effects Δ, while the latter defines an equation relating programs that have at least the effects Δ but potentially more. The □ modality is a *comonad*: the counit (extract below) witnesses that we can always transform

[30] The **constructor** keyword declares a function that we can call to construct an instance of a record; and that we can pattern match on to destruct record instances.

a lower bound on effects to an exact specification, by instantiating the extension witness with a proof of reflexivity.

```
extract : {P : Effect → Set₁} → □ P Δ → P Δ
extract px = □⟨ px ⟩ {{ ≲-refl }}
```

We can now redefine the *get-get* law such that it applies to all programs that have the State effect, but potentially other effects too.

```
get-get : □ Equation State
V   □⟨ get-get ⟩ = 1
Γ   □⟨ get-get ⟩ (A :: [])    = ℕ → ℕ → Free _ A
R   □⟨ get-get ⟩ (A :: [])    = A
lhs □⟨ get-get ⟩ (A :: []) k = `get >>= λ s → `get >>= λ s′ → k s s′
rhs □⟨ get-get ⟩ (A :: []) k = `get >>= λ s → k s s
```

The above definition of the *get-get* law now lets us prove the equality in Eq. (5.1); the term metavariable k ranges ranges over all continuations that return a tree of type Free Δ' A, for all Δ' such that State $\lesssim \Delta'$. This way, we can instantiate Δ' with an effect signature that subsumes both the State and the Throw, which in turn allows us to instantiate k with throw.

5.3 *Effect Theories*

Equations for an effect Δ can be combined into a *theory* for Δ. A theory for the effect Δ is simply a collection of equations, transformed using the □ modality to ensure that term metavariables can range over programs that include more effects than just Δ.

```
record Theory (Δ : Effect) : Set₁ where
  field
    arity     : Set
    equations : arity → □ Equation Δ
```

An effect theory consists of an arity, that defines the number of equations in the theory, and a function that maps arities to equations. We can think of effect theories as defining a specification for how implementations of an effect ought to behave. Although implementations may vary, depending for example on whether they are tailored to readability or efficiency, they should at least respect the equations of the theory of the effect they implement. We will make precise what it means for an implementation to respect an equation in Section 5.5.

Effect theories are closed under several composition operations that allow us to combine the equations of different theories into single theory. The most basic way of combining effect theories is by summing their arities.

```
_⟨+⟩_ : Theory Δ → Theory Δ → Theory Δ
arity     (T₁ ⟨+⟩ T₂) = arity T₁ ⊎ arity T₂
equations (T₁ ⟨+⟩ T₂) (inj₁ a) = equations T₁ a
equations (T₁ ⟨+⟩ T₂) (inj₂ a) = equations T₂ a
```

This way of combining effects is somewhat limiting, as it imposes that the theories we are combining are theories for the exact same effect. It is more likely, however, that we would want to combine theories for different effects. This requires that we can *weaken* effect theories with respect to the $\_\lesssim\_$ relation.

```
weaken-□ : {P : Effect → Set₁} → ⦃ Δ₁ ≲ Δ₂ ⦄ → □ P Δ₁ → □ P Δ₂
□⟨ weaken-□ ⦃ w ⦄ px ⟩ ⦃ w′ ⦄ = □⟨ px ⟩ ⦃ ≲-trans w w′ ⦄
```

```
weaken-theory : ⦃ Δ₁ ≲ Δ₂ ⦄ → Theory Δ₁ → Theory Δ₂
arity (weaken-theory T) = arity T
equations (weaken-theory T) = λ a → weaken-□ (T .equations a)
```

Categorically speaking, the observation that for a given effect-indexed type P we can transform a value of type $P\ \Delta_1$ to a value of type $P\ \Delta_2$ if we know that $\Delta_1 \lesssim \Delta_2$ is equivalent to saying that P is a functor from the category of containers and container morphisms to the category of sets. From this perspective, the existence of weakening for free Free, as witnessed by the $\sharp$ operation discussed in Section 3 implies that it too is a such a functor.

With weakening for theories at our disposal, we can combine effect theories for different effects into a theory of the coproduct of their respective effects. This requires us to first define appropriate witnesses relating coproducts to effect inclusion.

```
≲-⊕-left  : Δ₁ ≲ (Δ₁ ⊕ Δ₂)
≲-⊕-right : Δ₂ ≲ (Δ₁ ⊕ Δ₂)
```

It is now straightforward to show that effect theories are closed under the coproduct of effect signatures, by summing the weakened theories.

```
_[+]_ : Theory Δ₁ → Theory Δ₂ → Theory (Δ₁ ⊕ Δ₂)
T₁ [+] T₂ = weaken-theory ⦃ ≲-⊕-left ⦄ T₁ ⟨+⟩ weaken-theory ⦃ ≲-⊕-right ⦄ T₂
```

While this operation is in principle sufficient for our purposes, it forces a specific order on the effects of the combined theories. We can further generalize the operation above to allow for the effects of the combined theory to appear in any order. This requires the following instances.

```
≲-•-left  : ⦃ Δ₁ • Δ₂ ≈ Δ ⦄ → Δ₁ ≲ Δ
≲-•-right : ⦃ Δ₁ • Δ₂ ≈ Δ ⦄ → Δ₂ ≲ Δ
```

We show that effect theories are closed under coproducts up to reordering by, again, summing the weakened theories.

```
compose-theory : ⦃ Δ₁ • Δ₂ ≈ Δ ⦄ → Theory Δ₁ → Theory Δ₂ → Theory Δ
compose-theory T₁ T₂
  = weaken-theory ⦃ ≲-•-left ⦄ T₁ ⟨+⟩ weaken-theory ⦃ ≲-•-right ⦄ T₂
```

Since equations are defined by storing the syntax trees that define their left-hand and right-hand side, and effect trees are weakenable, we would expect equations to be weakenable too. Indeed, we can define the following function witnessing weakenability of equations.

```
weaken-eq : ⦃ Δ₁ ≲ Δ₂ ⦄ → Equation Δ₁ → Equation Δ₂
```

This begs the question: why would we opt to use weakenability of the □ modality (or, bother with the □ modality at all) to show that theories are weakenable, rather than using weaken-eq directly? Although the latter approach would indeed allow us to define the composition operations for effect theories defined above, the possible ways in which we can instantiate term metavariables remains too restrictive. That is, we would still not be able to prove the equality in Eq. (5.1), despite the fact that we can weaken the *get-get* law so that it applies to programs that use the Throw effect as well. Instantiations of the term metavariable k will be limited to weakened effect trees, precluding any instantiation that use operations of effects other than State, such as throw.

Finally, we define the following predicate to witness that an equation is part of a theory.

```
_◀_ : □ Equation Δ → Theory Δ → Set₁
eq ◀ T = ∃ λ a → T .equations a ≡ eq
```

We construct a proof eq ◀ T that an equation eq is part of a theory T by providing an arity together with a proof that T maps to eq for that arity.

5.4 Syntactic Equivalence of Effectful Programs

Propositional equality of effectful programs is too strict, as it precludes us from proving equalities that rely on a semantic understanding of the effects involved, such as the equality in Eq. (5.1). The solution is to define an inductive relation that captures syntactic equivalence modulo some effect theory. We base our definition of syntactic equality of effectful programs on the relation defining equivalent computations by Yang & Wu (2021), Definition 3.1, adapting their definition where necessary to account for the use of modal necessity in the definition of Theory.

```
data _≈⟨_⟩_ {Δ Δ′} ⦃ _ : Δ ≲ Δ′ ⦄
      : (m₁ : Free Δ′ A) → Theory Δ → (m₂ : Free Δ′ A) → Set₁ where
```

A value of type $m_1 \approx\langle\, T \,\rangle\, m_2$ witnesses that programs m_1 and m_2 are equal modulo the equations of theory T. The first three constructors ensure that it is an equivalence relation.

```
≈-refl  : m ≈⟨ T ⟩ m
≈-sym   : m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₁
≈-trans : m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₃ → m₁ ≈⟨ T ⟩ m₃
```

Then, we add the following congruence rule, which establishes that we can prove equality of two programs starting with the same operation by proving that the continuations yield equal programs for every possible value.

```
≈-cong :   (op : Op Δ′)
         → (k₁ k₂ : Ret Δ′ op → Free Δ′ A)
         → (∀ x → k₁ x ≈⟨ T ⟩ k₂ x)
         → impure (op , k₁) ≈⟨ T ⟩ impure (op , k₂)
```

The final constructor allows to prove equality of programs by reifying equations of an effect theory.

```
≈-eq :   (eq : □ Equation Δ)
       → (px : eq ◀ T)
       → (vs : Vec Set (V (□⟨ eq ⟩)))
       → (γ : Γ (□⟨ eq ⟩) vs)
       → (k : R (□⟨ eq ⟩) vs → Free Δ′ A)
       → (lhs (□⟨ eq ⟩) vs γ >>= k) ≈⟨ T ⟩ (rhs (□⟨ eq ⟩) vs γ >>= k)
```

Since the equations of a theory are wrapped in the □ modality, we cannot refer to its components directly, but we must first provide a suitable extension witness.

With the ≈-eq constructor, we can prove equivalence between the left-hand and right-hand side of an equation, sequenced with an arbitrary continuation k. For convenience, we define the following lemma that allows us to apply an equation where the sides of the equation do not have a continuation.

```
use-equation :   ⦃ _ : Δ ≲ Δ′ ⦄
               → {T : Theory Δ}
               → (eq : □ Equation Δ)
               → eq ◀ T
               → (vs : Vec Set (V □⟨ eq ⟩))
               → {γ : Γ (□⟨ eq ⟩) vs}
               → lhs (□⟨ eq ⟩) vs γ ≈⟨ T ⟩ rhs (□⟨ eq ⟩) vs γ
```

The definition of use-equation follows readily from the right-identity law for monads, i.e., $m \gg\!= \text{pure} \equiv m$, which allows us to instantiate ≈-eq with pure.

To construct proofs of equality it is convenient to use the following set of combinators to write proof terms in an equational style. They are completely analogous to the combinators commonly used to construct proofs of Agda's propositional equality, for example, as found in PLFA (Wadler *et al.*, 2020).

```
module ≈-Reasoning (T : Theory Δ) ⦃ _ : Δ ≲ Δ′ ⦄ where
  begin_ : {m₁ m₂ : Free Δ′ A} → m₁ ≈⟨ T ⟩ m₂ → m₁ ≈⟨ T ⟩ m₂
  begin eq = eq

  _■ : (m : Free Δ′ A) → m ≈⟨ T ⟩ m
  m ■ = ≈-refl

  _≈⟪⟫_ : (m₁ : Free Δ′ A) {m₂ : Free Δ′ A} → m₁ ≈⟨ T ⟩ m₂ → m₁ ≈⟨ T ⟩ m₂
  m₁ ≈⟪⟫ eq = eq

  _≈⟪_⟫_ : (m₁ {m₂ m₃} : Free Δ′ A) → m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₃ → m₁ ≈⟨ T ⟩ m₃
  m₁ ≈⟪ eq₁ ⟫ eq₂ = ≈-trans eq₁ eq₂
```

We now have all the necessary tools to prove syntactic equality of programs modulo a theory of their effect. To illustrate, we consider how to prove the equation in Eq. (5.1). First, we define a theory for the State effect containing the get-get◀ law. While this is not the only law typically associated with State, for this example it is enough to only have the get-get law.

```
StateTheory : Theory State
arity StateTheory           = ⊤
equations StateTheory tt = get-get
```

Now to prove the equality in Eq. (5.1) is simply a matter of invoking the get-get law.

```
get-get-throw :
    ⦃ _ : Throw ≲ Δ ⦄ ⦃ _ : State ≲ Δ ⦄
  → (‘get ≫= λ s → ‘get ≫= λ s′ → ‘throw {A = A})
    ≈⟨ StateTheory ⟩ (‘get ≫= λ s → ‘throw)
get-get-throw {A = A} = begin
   ‘get ≫= (λ s → ‘get ≫= (λ s′ → ‘throw))
  ≈⟪ use-equation get-get (tt , refl) (A :: []) ⟫
   ‘get ≫= (λ s → ‘throw)
  ■
  where open ≈-Reasoning StateTheory
```

5.5 Handler Correctness

A handler is correct with respect to a given theory if handling syntactically equal programs yields equal results. Since handlers are defined as algebras over effect signatures, we start by defining what it means for an algebra of an effect Δ to respect an equation of the same effect, adapting Definition 2.1 from the exposition of Yang & Wu (2021).

```
Respects : Alg Δ A → Equation Δ → Set₁
Respects alg eq = ∀ {vs γ k} →
  fold k alg (lhs eq vs γ) ≡ fold k alg (rhs eq vs γ)
```

An algebra *alg* respects an equation *eq* if folding with that algebra produces propositionally equal results for the left- and right-hand side of the equation, for all possible instantiations of its type and term metavariables, and continuations *k*.

A handler *H* is correct with respect to a given theory *T* if its algebra respects all equations of *T* (Yang & Wu, 2021, Definition 4.3).

```
Correct : {P : Set} → Theory Δ → ⟨ A ! Δ ⇒ P ⇒ B ! Δ′ ⟩ → Set₁
Correct T H = ∀ {eq} → eq ◂ T → Respects (H .hdl) (extract eq)
```

We can now show that the handler for the State effect defined in Fig. 1 is correct with respect to StateTheory. The proof follows immediately by reflexivity.

```
hStCorrect : Correct {A = A} {Δ′ = Δ} StateTheory hSt
hStCorrect (tt , refl) {_ :: []} {γ = k} = refl
```

5.6 Theories of Higher-Order Effects

For the most part, equations and theories for higher-order effects are defined in the same way as for first-order effects and support many of the same operations. Indeed, the definition of equations ranging over higher-order effects is exactly the same as its first-order counterpart, the most major difference being that the left-hand and right-hand side are now defined as Hefty trees. To ensure compatibility with the use of type universes to avoid size-issues, we must also allow type metavariables to range over the types in a universe in addition to Set. For this reason, the set of type metavariables is no longer described by a natural number, but rather by a list of kinds, which stores for each type metavariable whether it ranges over a types in a universe, or an Agda Set.

```
data Kind : Set where set type : Kind
```

A TypeContext carries unapplied substitutions for a given set of type metavariables, and is defined by induction over a list of kinds.[31]

```
TypeContext : List Kind → Set₁
TypeContext []            = Level.Lift _ ⊤
TypeContext (set  :: vs) = Set × TypeContext vs
TypeContext (type :: vs) = Level.Lift (sℓ 0ℓ) Type × TypeContext vs
```

Equations of higher-order effects are then defined as follows.

```
record Equationᴴ (H : Effectᴴ) : Set₁ where
  field
    V       : List Kind
    Γ       : TypeContext V → Set
    R       : TypeContext V → Set
    lhs rhs : (vs : TypeContext V) → Γ vs → Hefty H (R vs)
```

This definition of equations suffers the same problem when it comes to term metavariables, which here too can only range over programs that exhibit the exact effect that the equation is defined for. Again, we address the issue using an embedding of modal necessity to close over all possible extensions of this effect. The definition is analogous to the one in Section 5.2, but this time we use higher-order effect subtyping as the modal accessibility relation:

```
record □ (P : Effectᴴ → Set₁) (H : Effectᴴ) : Set₁ where
  constructor necessary
  field □⟨_⟩ : ∀ {H′} → ⦃ H ≲ᴴ H′ ⦄ → P H′
```

To illustrate: we can define the *catch-return* law from the introduction of this section as a value of type □ Equationᴴ Catch a follows. Since the ‵catch operation relies on a type universe to avoid size issues, the sole type metavariable of this equation must range over the types in this universe as well.

```
catch-return : □ Equationᴴ Catch
V   □⟨ catch-return ⟩          = type :: []
```

[31] Level.Lift lifts a type in Set to a type in Set_1. The constructor of Level.Lift is lift.

```
Γ   □⟨ catch-return ⟩ (lift t , _) = ⟦ t ⟧^T × Hefty _ ⟦ t ⟧^T
R   □⟨ catch-return ⟩ (lift t , _) = ⟦ t ⟧^T
lhs □⟨ catch-return ⟩ _ (x , m)  = `catch (pure x) m
rhs □⟨ catch-return ⟩ _ (x , m)  = pure x
```

Theories of higher-order effects bundle extensible equations. The setup is the same as for theories of first-order effects.

```
record Theory^H (H : Effect^H) : Set1 where
  field
    arity : Set
    equations : arity → □ Equation^H H
```

The following predicate establishes that an equation is part of a theory. We prove this fact by providing an arity whose corresponding equation is equal to *eq*.

```
_◂^H_ : □ Equation^H H → Theory^H H → Set1
eq ◂^H Th = ∃ λ a → eq ≡ equations Th a
```

Weakenability of theories of higher-order effects then follows from weakenability of its equations.

```
weaken-□ : ∀ {P} → ⦃ H1 ≲^H H2 ⦄ → □ P H1 → □ P H2
□⟨ weaken-□ ⦃ w ⦄ px ⟩ ⦃ w′ ⦄ = □⟨ px ⟩ ⦃ ≲^H-trans w w′ ⦄

weaken-theory^H : ⦃ H1 ≲^H H2 ⦄ → Theory^H H1 → Theory^H H2
arity         (weaken-theory^H Th)   = Th .arity
equations (weaken-theory^H Th) a = weaken-□ (Th .equations a)
```

Theories of higher-order effects can be combined using the following sum operation. The resulting theory contains all equations of both argument theories.

```
_⟨+⟩^H_ : ∀[ Theory^H ⇒ Theory^H ⇒ Theory^H ]
arity         (Th1 ⟨+⟩^H Th2)           = arity Th1 ⊎ arity Th2
equations (Th1 ⟨+⟩^H Th2) (inj1 a) = equations Th1 a
equations (Th1 ⟨+⟩^H Th2) (inj2 a) = equations Th2 a
```

Theories of higher-order effects are closed under sums of higher-order effect theories as well. This operation is defined by appropriately weakening the respective theories, for which we need the following lemmas witnessing that higher-order effect signatures can be injected in a sum of signatures.

```
≲-∔-left   : H1 ≲^H (H1 ∔ H2)
≲-∔-right : H2 ≲^H (H1 ∔ H2)
```

The operation that combines theories under signature sums is then defined like so.

```
_[+]^H_ : Theory^H H1 → Theory^H H2 → Theory^H (H1 ∔ H2)
Th1 [+]^H Th2
  = weaken-theory^H ⦃ ≲-∔-left ⦄ Th1 ⟨+⟩^H weaken-theory^H ⦃ ≲-∔-right ⦄ Th2
```

5.7 Equivalence of Programs with Higher-Order Effects

We define the following inductive relation to capture equivalence of programs with higher-order effects modulo the equations of a given theory.

```
data _≅⟨_⟩_ ⦃ _ : H₁ ≲ᴴ H₂ ⦄
     : (m₁ : Hefty H₂ A) → Theoryᴴ H₁ → (m₂ : Hefty H₂ A) → Set₁ where
```

To ensure that it is indeed an equivalence relation, we include constructors for reflexivity, symmetry, and transitivity.

```
  ≅-refl  : ∀ {m : Hefty H₂ A}
            → m ≅⟨ Th ⟩ m

  ≅-sym   : ∀ {m₁ : Hefty H₂ A} {m₂}
            → m₁ ≅⟨ Th ⟩ m₂
            → m₂ ≅⟨ Th ⟩ m₁

  ≅-trans : ∀ {m₁ : Hefty H₂ A} {m₂ m₃}
            → m₁ ≅⟨ Th ⟩ m₂ → m₂ ≅⟨ Th ⟩ m₃
            → m₁ ≅⟨ Th ⟩ m₃
```

Furthermore, we include the following congruence rule that equates two program trees that have the same operation at the root, if their continuations are equivalent for all inputs.

```
  ≅-cong :   (op : Opᴴ H₂)
           → (k₁ k₂ : Retᴴ H₂ op → Hefty H₂ A)
           → (s₁ s₂ : (ψ : Fork H₂ op) → Hefty H₂ (Ty H₂ ψ))
           → (∀ {x} → k₁ x ≅⟨ Th ⟩ k₂ x)
           → (∀ {ψ} → s₁ ψ ≅⟨ Th ⟩ s₂ ψ)
           → impure (op , k₁ , s₁) ≅⟨ Th ⟩ impure ( op , k₂ , s₂ )
```

Finally, we include a constructor that equates two programs using an equation of the theory.

```
  ≅-eq   :   (eq : □ Equationᴴ H₁)
           → eq ◄ᴴ Th
           → (vs : TypeContext (V □⟨ eq ⟩))
           → (γ : Γ □⟨ eq ⟩ vs)
           → (k : R □⟨ eq ⟩ vs → Hefty H₂ A)
           → (lhs □⟨ eq ⟩ vs γ >>= k) ≅⟨ Th ⟩ (rhs □⟨ eq ⟩ vs γ >>= k)
```

We can define the same reasoning combinators as in Section 5.4 to construct proofs of equivalence for programs with higher-order effects.

```
module ≅-Reasoning ⦃ _ : H₁ ≲ᴴ H₂ ⦄ (Th : Theoryᴴ H₁) where

  begin_ : {m₁ m₂ : Hefty H₂ A} → m₁ ≅⟨ Th ⟩ m₂ → m₁ ≅⟨ Th ⟩ m₂
  begin eq = eq

  _■ : (c : Hefty H₂ A) → c ≅⟨ Th ⟩ c
```

```
  c ■ = ≅-refl

  _≅⟨⟩_ : (m₁ : Hefty H₂ A) {m₂ : Hefty H₂ A} → m₁ ≅⟨ Th ⟩ m₂ → m₁ ≅⟨ Th ⟩ m₂
  c₁ ≅⟨⟩ eq = eq

  _≅⟨_⟩_ : (c₁ {c₂ c₃} : Hefty H₂ A) → c₁ ≅⟨ Th ⟩ c₂ → c₂ ≅⟨ Th ⟩ c₃ → c₁ ≅⟨ Th ⟩ c₃
  c₁ ≅⟨ eq₁ ⟩ eq₂ = ≅-trans eq₁ eq₂
```

To illustrate, we can prove that the programs catch throw (censor f m) and censor f m are equal under a theory for the *afCatch* effect that contains the *catch-return* law.

```
catch-return-censor : ∀ {t : Type} {f} {x : ⟦ t ⟧ᵀ} {m : Hefty H ⟦ t ⟧ᵀ}
                    → ⦃ _ : Catch ≲ᴴ H ⦄ → ⦃ _ : Censor ≲ᴴ H ⦄
                    → ‵catch (pure x) (‵censor f m)
                      ≅⟨ CatchTheory ⟩ pure x
catch-return-censor {f = f} {x = x} {m = m} =
  begin
    ‵catch (pure x) (‵censor f m)
  ≅⟨ use-equationᴴ catch-return (tt , refl) _ ⟩
    pure x
  ■
  where open ≅-Reasoning _
```

The equivalence proof above makes, again, essential use of modal necessity. That is, by closing over all possible extensions of the Catch effe, the term metavariable in the *catch-return* law to range over programs that have higher-order effects other than Catch, which is needed to apply the law if the second branch of the catch operation contains the censor operation.

5.8 Correctness of Elaborations

As the first step towards defining correctness of elaborations, we must specify what it means for an algebra over a higher-order effect signature H to respect an equation. The definition is broadly similar to its counterpart for first-order effects in Section 5.5, with the crucial difference that the definition of "being equation respecting" for algebras over higher-order effect signatures is parameterized over a binary relation $\_\approx\_$ between first-order effect trees. In practice, this binary relation will be instantiated with the inductive equivalence relation defined in Section 5.4; propositional equality would be too restrictive, since that does not allow us prove equivalence of programs using equations of the first-order effect(s) that we elaborate into.

```
Respectsᴴ : (_≈_ : ∀ {A} → Free Δ A → Free Δ A → Set₁)
          → Algᴴ H (Free Δ) → Equationᴴ H → Set₁
Respectsᴴ _≈_ alg eq =
  ∀ {vs γ} → cataᴴ pure alg (lhs eq vs γ) ≈ cataᴴ pure alg (rhs eq vs γ)
```

Since elaborations are composed in parallel, the use of necessity in the definition of equations has additional consequences for the definiton of elaboration correctness. That is, correctness of an elaboration is defined with respect to a theory whose equations have left-hand and right-hand sides that may contain term metavariables that range over programs with more higher-order effects than those the elaboration is defined for. Therefore, to state correctness, we must also close over all possible ways these additional effects are elaborated. For this, we define the following binary relation on extensible elaborations.[32]

```
record _⊑_ (e₁ : □ (Elaboration H₁) Δ₁) (e₂ : □ (Elaboration H₂) Δ₂) : Set₁ where
  field
    ⦃ ≲-eff   ⦄ : Δ₁ ≲ Δ₂
    ⦃ ≲ᴴ-eff ⦄ : H₁ ≲ᴴ H₂
    preserves-cases
      : ∀ {M} (m : ⟦ H₁ ⟧ᴴ M A)
      → (e′ : ∀[ M ⇒ Free Δ₂ ])
      →  □⟨ e₁ ⟩ .alg (map-sigᴴ (λ {x} → e′ {x}) m)
        ≡ extract e₂ .alg (map-sigᴴ (λ {x} → e′ {x}) (injᴴ {X = A} m))
```

A proof of the form $e_1 \sqsubseteq e_2$ witnesses that the elaboration e_1 is included in e_2. Informally, this means that e_2 may elaborate a bigger set of higher-order effects, for which it may need to refer to a bigger set of first-order effects, but for those higher-order effects that both e_1 and e_2 know how to elaborate, they should agree on how those effects are elaborated.

We then define correctness of elaborations as follows.

```
Correctᴴ : Theoryᴴ H → Theory Δ → □ (Elaboration H) Δ → Set₁
Correctᴴ Th T e =
  ∀ {Δ′ H′}
  → (e′ : □ (Elaboration H′) Δ′)
  → ⦃ _ : e ⊑ e′ ⦄
  → {eq : □ Equationᴴ _}
  → eq ◂ᴴ Th
  → Respectsᴴ (_≈⟨ T ⟩_) (extract e′) □⟨ eq ⟩
```

Which is to say that an elaboration is correct with respect to a theory of the higher-order effects it elaborates (*Th*) and a theory of the first-order effects it elaborates into (*T*), if all possible extensions of said elaboration respect all equations of the higher-order theory, modulo the equations of the first-order theory.

Crucially, correctness of elaborations is preserved under composition of elaborations. Fig. 8 shows the type of the corresponding correctness theorem in Agda; for the full details of the proof we refer to the Agda formalization accompanying this paper (van der Rest & Poulsen, 2024). We remark that correctness of a composed elaboration is defined with respect to the composition of the theories of the first-order effects that the respective elaborations use. Constructing a handler that is correct with respect to this composed first-order effect theory is a separate concern; a solution based on *fusion* is detailed in the work by Yang & Wu (2021).

[32] Here, $\mathsf{inj}^{\mathsf{H}}$ is the higher-order counterpart to the inj function discussed in Section 2.2.

```
compose-elab-correct : ⦃ _ : Δ₁ ● Δ₂ ≈ Δ ⦄
                     → (e₁ : □ (Elaboration H₁) Δ₁)
                     → (e₂ : □ (Elaboration H₂) Δ₂)
                     → (T₁ : Theory Δ₁)
                     → (T₂ : Theory Δ₂)
                     → (Th₁ : Theoryᴴ H₁)
                     → (Th₂ : Theoryᴴ H₂)
                     → Correctᴴ Th₁ T₁ e₁
                     → Correctᴴ Th₂ T₂ e₂
                     → Correctᴴ (Th₁ [+]ᴴ Th₂) (compose-theory T₁ T₂)
                         (compose-elab e₁ e₂)
```

Fig. 8. The central correctness theorem, which establishes that correctness of elaborations is preserved under composition.

5.9 Proving Correctness of Elaborations

To illustrate how the reasoning infrastructure build up in this section can be applied to verify correctness of elaborations, we show how to verify the *catch-return* law for the elaboration eCatch defined in Section 3.4. First, we define the following syntax for invoking a known elaboration.

```
module Elab (e : □ (Elaboration H) Δ) where
  ℰ⟦_⟧ : Hefty H A → Free Δ A
  ℰ⟦ m ⟧ = elaborate (extract e) m
```

When opening the module *Elab*, we can use the syntax ℰ⟦ m⟧ for elaborating m with some known elaboration, which helps to simplify and improve readability of equational proofs for higher-order effects.

Now, to prove that eCatch is correct with respect to a higher-order theory for the Catch effect containing the *catch-return* law, we must produce a proof that the programs ℰ⟦ ‵catch (return x) m ⟧ and ℰ⟦ return ⟧ are equal (in the sense of the inductive equivalence relation defined in Section 5.4) with respect to some first-order theory for the Throw effect. In this instance, we do not need any equations from this underlying theory to prove the equality, but sometimes it is necessary to invoke equations of the underlying first-order effects to prove correctness of an elaboration.

```
eCatchCorrect : {T : Theory Throw} → Correctᴴ CatchTheory T eCatch
eCatchCorrect {Δ′ = Δ′} e′ ⦃ ζ ⦄ (tt , refl) {γ = x , m} =
  begin
    ℰ⟦ ‵catch (pure x) m ⟧
  ≈⟪ from-≡ (sym $ ζ .preserves-cases _ ℰ⟦_⟧) ⟫
    (♯ (given hThrow handle (pure x) $ tt)) ≫= maybe′ pure (ℰ⟦ m ⟧)
  ≈⟪⟫ {- By definition of hThrow -}
    (pure (just x) ≫= maybe′ pure ((ℰ⟦ m ⟧ ≫= pure)))
  ≈⟪⟫ {- By definition of ≫= -}
```

Effect	Laws	
Throw	$\text{`throw} \gg= k \equiv k$	*bind-throw*
State	$\text{`get} \gg= \lambda\, s \to \text{`get} \gg= k\, s \equiv \text{`get} \gg= k\, s\, s$	*get-get*
	$\text{`get} \gg= \text{`put} \equiv \text{pure}\; x$	*get-put*
	$\text{`put}\; s \gg \text{`get} \equiv \text{`put}\; s \gg \text{pure}\; s$	*put-get*
	$\text{`put}\; s \gg \text{`put}\; s' \equiv \text{`put}\; s'$	*put-put*
Reader	$\text{`ask} \gg m \equiv m$	*ask-query*
	$\text{`ask} \gg= \lambda\, r \to \text{`ask} \gg= k\, r \equiv \text{`ask} \gg= \lambda\, r \to k\, r\, r$	*ask-ask*
	$m \gg= \lambda\, x \to \text{`ask} \gg= \lambda\, r \to k\, x\, r \equiv \text{`ask} \gg= \lambda\, r \to m \gg= \lambda\, x \to k\, x\, r$	*ask-bind*
LocalReader	$\text{`local}\; f\; (\text{pure}\; x) \equiv \text{pure}\; x$	*local-pure*
	$\text{`local}\; f\; (m \gg= k) \equiv \text{`local}\; f\; m \gg= \text{`local}\; f \circ k$	*local-bind*
	$\text{`local f `ask} \equiv \text{pure} \circ f$	*local-ask*
	$\text{`local}\; (f \circ g)\; m \equiv \text{`local}\; g\; (\text{`local}\; f\; m)$	*local-local*
Catch	$\text{`catch}\; (\text{pure}\; x)\; m \equiv \text{pure}\; x$	*catch-pure*
	$\text{`catch `throw}\; m \equiv m$	*catch-throw*$_1$
	$\text{`catch}\; m\; \text{`throw} \equiv m$	*catch-throw*$_2$
Lambda	$\text{`abs}\; f \gg= \lambda\, f' \to \text{`app}\; f'\; m \equiv m \gg= f$	*beta*
	$\text{pure}\; f \equiv \text{`abs}\; (\lambda\, x \to \text{`app}\; f\; (\text{`var}\; x))$	*eta*

Table 1. Overview of effects, their operations, and verified laws in the Agda code.

```
  ℰ⟦ pure x ⟧
■
where
  open ≈-Reasoning _
  open Elab e′
```

In the Agda formalization accompanying this paper (van der Rest & Poulsen, 2024), we verify correctness of elaborations for the higher-order operations that are part of the 3MT library by Delaware *et al.* (2013). Table 1 shows an overview of first-order and higher-order effects included in the development, and the laws which we prove about their handlers respectively elaborations.

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 (1989*a*) 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 (Steele Jr., 1994; Jones & Duponcheel, 1993; Filinski, 1999; Cenciarelli & Moggi, 1993). Of these solutions, monad transformers (Cenciarelli & Moggi, 1993; Liang *et al.*, 1995; Jaskelioff, 2008) is the more widely adopted solution. However, more recently, algebraic effects (Plotkin & 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 & 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 & Ishii, 2015), or Idris' `Effects` DSL (Brady, 2013*b*); but also standalone languages such as Eff (Bauer & Pretnar, 2015), Koka (Leijen, 2017), Frank (Lindley *et al.*, 2017), and Effekt (Brachthäuser *et al.*, 2020).[33]

As discussed in Section 1.2 and Section 2.5, some modularity benefits of algebraic effects and handlers do not carry over to higher-order effects. Scoped effects and handlers (Wu *et al.*, 2014; Piróg *et al.*, 2018; Yang *et al.*, 2022) address this shortcoming for *scoped operations*, as we summarized in Section 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 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.

Another recent paper by van den Berg & Schrijvers (2023) presents a unified framework for describing higher-order effects, which can be specialized to recover several instances such as Scoped Effects (Wu *et al.*, 2014) or Latent Effects (van den Berg *et al.*, 2021). They present a generic free monad generated from higher-order signatures that coincides with the type of Hefty trees that we present in Section 3. Their approach relies on a *Generalized Fold* (Bird & Paterson, 1999) for describing semantics of handling operations, in contrast

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

to the approach in this paper, where we adopt a two-stage process of elaboration and handling that can be expressed using the standard folds of first-order and higher-order free monads. To explore how the use of generalized folds versus standard folds affects the relative expressivity of approaches to higher-order effects is a subject of further study.

The equational framework we present in Section 5 is inspired by the work of Yang & Wu (2021). Specifically, the notion of higher-order effect theory we formalized in Agda is an extension of the notion of (first-order) effect theory they use. In closely related recent work by Kidney *et al.* (2024), they present a formalization of first-order effect theories in *Cubical Agda* (Vezzosi *et al.*, 2021). Whereas our formalization requires extrinsic verification of the equalities of an effect theory, they use *quotient types* as provided by homotopy type theory (Program, 2013) and cubical type theory (Angiuli *et al.*, 2021; Cohen *et al.*, 2017) to verify that handlers intrinsically respect their effect theories. They also present a Hoare logic for verifying pre- and post-conditions. An interesting question for future work is whether this logic and the framework of Kidney *et al.* (2024) could be extended to higher-order effect theories.

In other recent work, Lindley *et al.* (2024) developed an equational reasoning system for scoped effects. The system is based on so-called *parameterized algebraic theories*; i.e., effect theories with two kinds of variables: one for values, and one for computations representing *scopes*. They demonstrate how their framework supports key examples from the literature: nondeterminism with semi-determinism, catching exceptions, and local state. The framework we present in Section 5 supports variables ranging over either values or computations (see, e.g., catch-return in Section 5.6). Our framework does not explicitly distinguish these two kinds of variables. We demonstrate that our approach lets us verify the laws of the higher-order exception catching effect (Section 5.9), and characterize the semantics of composing higher-order effect theories (Section 5.8). Key to our approach is that the correctness of elaborations is with respect to an algebraic effect theory. If this underlying theory encodes a scoped syntax, we may need parameterized algebraic effect theories à la Lindley *et al.* (2024) to properly characterize it.

The elaboration semantics of hefty algebras that we defined in Section 3 is based on *initial algebra semantics*—that is, it is given by a fold over an inductively defined syntax tree. An alternative approach is Wand (1979) calls *final algebra semantics*, popularly known as *final encodings* Kamin (1983) or *finally tagless style* (Carette *et al.*, 2009). Here, the idea is that, instead of declaring syntax as an inductive datatype, we declare it as a record type. For example, consider the following record type:

```
record Symantics (Repr : Set → Set) : Set₁ where
  field num : ℕ → Repr ℕ
        lam : (Repr A → Repr B) → Repr (A → B)
        app : Repr (A → B) → Repr A → Repr B
```

Following Carette *et al.* (2009), this record is called Symantics because its interface gives the syntax of the object language and its instances give the semantics. For example:

```
SetSymantics : Symantics id
num SetSymantics = id
```

```
lam  SetSymantics = id
app  SetSymantics = _$_
```

A benefit of this approach is that it yields programs that can be executed more efficiently, because compilers can more readily optimize programs given by a concrete record instance than programs given by an inductive data type and a fold over it. These benefits extend to effects. Devriese (2019) presents a final tagless encoding of monads in Haskell, using dictionary passing. We expect that it is possible to encode modular elaborations of higher-order effects in a similar final style; i.e., by programming against records that encode a higher-order interface, and whose implementation is given by a free monad. This final encoding should be semantically equivalent to initial encoding presented in this paper.

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 & 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 modular 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 portable 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.

Acknowledgements 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

Abbott, M. G., Altenkirch, T. and Ghani, N. (2003) Categories of containers. Gordon, A. D. (ed), *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 2620, pp. 23–38. Springer.

Abbott, M. G., Altenkirch, T. and Ghani, N. (2005) Containers: Constructing strictly positive types. *Theor. Comput. Sci.* **342**(1):3–27.

Allais, G., Atkey, R., Chapman, J., McBride, C. and McKinna, J. (2021) A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.* **31**:e22.

Angiuli, C., Brunerie, G., Coquand, T., Harper, R., (Favonia), K. H. and Licata, D. R. (2021) Syntax and models of cartesian cubical type theory. *Math. Struct. Comput. Sci.* **31**(4):424–468.

Arbib, M. A. and Manes, E. G. (1975) *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press.

Awodey, S. (2010) *Category Theory*. 2nd edn. Oxford University Press, Inc.

Bauer, A. and Pretnar, M. (2015) Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* **84**(1):108–123.

Biernacki, D., Piróg, M., Polesiuk, P. and Sieczkowski, F. (2018) Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* **2**(POPL):8:1–8:30.

Bird, R. S. and Paterson, R. (1999) Generalised folds for nested datatypes. *Formal Aspects Comput.* **11**(2):200–222.

Brachthäuser, J. I., Schuster, P. and Ostermann, K. (2020) Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* **4**(OOPSLA):126:1–126:30.

Brady, E. C. (2013a) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(5):552–593.

Brady, E. C. (2013b) Programming and reasoning with algebraic effects and dependent types. *In:* Morrisett & Uustalu (2013).

Carette, J., Kiselyov, O. and Shan, C. (2009) Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5):509–543.

Castagna, G. and Gordon, A. D. (eds). (2017) *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM.

Cenciarelli, P. and Moggi, E. (1993) A syntactic approach to modularity in denotational semantics.

Claessen, K. (1999) A poor man's concurrency monad. *J. Funct. Program.* **9**(3):313–323.

Cohen, C., Coquand, T., Huber, S. and Mörtberg, A. (2017) Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP* **4**(10):3127–3170.

Delaware, B., Keuchel, S., Schrijvers, T. and d. S. Oliveira, B. C. (2013) Modular monadic meta-theory 319–330.

Devriese, D. (2019) Modular effects in haskell through effect polymorphism and explicit dictionary applications: a new approach and the μverifast verifier as a case study. *In:* Eisenberg (2019).

Eisenberg, R. A. (ed). (2019) *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. ACM.

Filinski, A. (1999) Representing layered monads. Appel, A. W. and Aiken, A. (eds), *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999* pp. 175–188. ACM.

Fiore, M. P. and Staton, S. (2014) Substitution, jumps, and algebraic effects. Henzinger, T. A. and Miller, D. (eds), *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* pp. 41:1–41:10. ACM.

Hancock, P. G. and Setzer, A. (2000) Interactive programs in dependent type theory. Clote, P. and Schwichtenberg, H. (eds), *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings*. Lecture Notes in Computer Science 1862, pp. 317–331. Springer.

Hyland, M., Plotkin, G. D. and Power, J. (2006) Combining effects: Sum and tensor. *Theor. Comput. Sci.* **357**(1-3):70–99.

Jaskelioff, M. (2008) Monatron: An extensible monad transformer library. Scholz, S. and Chitil, O. (eds), *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 5836, pp. 233–248. Springer.

Jones, M. P. (1995) Functional programming with overloading and higher-order polymorphism. Jeuring, J. and Meijer, E. (eds), *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 925, pp. 97–136. Springer.

Jones, M. P. and Duponcheel, L. (1993) *Composing Monads*. Research Report YALEU/DCS/RR-1004. Yale University, New Haven, Connecticut, USA.

Kamin, S. N. (1983) Final data types and their specification. *ACM Trans. Program. Lang. Syst.* **5**(1):97–123.

Kammar, O., Lindley, S. and Oury, N. (2013) Handlers in action. *In:* Morrisett & Uustalu (2013).

Kidney, D. O., Yang, Z. and Wu, N. (2024) Algebraic effects meet Hoare logic in Cubical Agda. *Proc. ACM Program. Lang.* **8**(POPL):1663–1695.

Kiselyov, O. and Ishii, H. (2015) Freer monads, more extensible effects. Lippmeier, B. (ed), *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015* pp. 94–105. ACM.

Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. *In:* Castagna & Gordon (2017).

Levy, P. B. (2006) Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.* **19**(4):377–414.

Liang, S., Hudak, P. and Jones, M. P. (1995) Monad transformers and modular interpreters. Cytron, R. K. and Lee, P. (eds), *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995* pp. 333–343. ACM Press.

Lindley, S., McBride, C. and McLaughlin, C. (2017) Do be do be do. *In:* Castagna & Gordon (2017).

Lindley, S., Matache, C., Moss, S. K., Staton, S., Wu, N. and Yang, Z. (2024) Scoped effects as parameterized algebraic theories. Weirich, S. (ed), *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*. Lecture Notes in Computer Science 14576, pp. 3–21. Springer.

Martin-Löf, P. (1984) *Intuitionistic type theory*. Studies in proof theory, vol. 1. Bibliopolis.

Meijer, E., Fokkinga, M. M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. Hughes, J. (ed), *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Lecture Notes in Computer Science 523, pp. 124–144. Springer.

Moggi, E. (1989a) *An Abstract View of Programming Languages*. Tech. rept. ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.

Moggi, E. (1989b) Computational lambda-calculus and monads. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989* pp. 14–23. IEEE Computer Society.

Morris, J. G. and McKinna, J. (2019) Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* **3**(POPL):12:1–12:28.

Morrisett, G. and Uustalu, T. (eds). (2013) *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM.

Mosses, P. D. (2004) Modular structural operational semantics. *J. Log. Algebraic Methods Program.* **60-61**:195–228.

Pfenning, F. and Elliott, C. (1988) Higher-order abstract syntax. Wexelblat, R. L. (ed), *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*

(PLDI), Atlanta, Georgia, USA, June 22-24, 1988 pp. 199–208. ACM.

Pierce, B. C. (1991) *Basic category theory for computer scientists*. Foundations of computing. MIT Press.

Piróg, M. and Gibbons, J. (2014) The coinductive resumption monad. Jacobs, B., Silva, A. and Staton, S. (eds), *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 308, pp. 273–288. Elsevier.

Piróg, M., Schrijvers, T., Wu, N. and Jaskelioff, M. (2018) Syntax and semantics for operations with scopes. Dawar, A. and Grädel, E. (eds), *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018* pp. 809–818. ACM.

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

Plotkin, G. D. and Power, J. (2002) Notions of computation determine monads. Nielsen, M. and Engberg, U. (eds), *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 2303, pp. 342–356. Springer.

Plotkin, G. D. and Power, J. (2003) Algebraic operations and generic effects. *Appl. Categorical Struct.* **11**(1):69–94.

Plotkin, G. D. and Pretnar, M. (2009) Handlers of algebraic effects. Castagna, G. (ed), *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 5502, pp. 80–94. Springer.

Poulsen, C. B. and van der Rest, C. (2023) Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.* **7**(POPL):1801–1831.

Pretnar, M. (2015) An introduction to algebraic effects and handlers. invited tutorial paper. Ghica, D. R. (ed), *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 319, pp. 19–35. Elsevier.

Program, T. U. F. (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Rosu, G. and Serbanuta, T. (2010) An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* **79**(6):397–434.

Schmidt, D. (1986) *Denotational Semantics*. Allyn and Bacon.

Schrijvers, T., Wu, N., Desouter, B. and Demoen, B. (2014) Heuristics entwined with handlers combined: From functional specification to logic programming implementation. Chitil, O., King, A. and Danvy, O. (eds), *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014* pp. 259–270. ACM.

Schrijvers, T., Piróg, M., Wu, N. and Jaskelioff, M. (2019) Monad transformers and modular algebraic effects: what binds them together. *In:* Eisenberg (2019).

Sculthorpe, N., Torrini, P. and Mosses, P. D. (2015) A modular structural operational semantics for delimited continuations. Danvy, O. and de'Liguoro, U. (eds), *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015*. EPTCS 212, pp. 63–80.

Steele Jr., G. L. (1994) Building interpreters by composing monads. Boehm, H., Lang, B. and Yellin, D. M. (eds), *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994* pp. 472–492. ACM Press.

Swierstra, W. (2008) Data types à la carte. *J. Funct. Program.* **18**(4):423–436.

Taha, W. and Sheard, T. (2000) Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* **248**(1-2):211–242.

Thielecke, H. (1997) *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh.

van den Berg, B. and Schrijvers, T. (2023) A framework for higher-order effects & handlers. *CoRR* **abs/2302.01415**.

van den Berg, B., Schrijvers, T., Poulsen, C. B. and Wu, N. (2021) Latent effects for reusable language components. Oh, H. (ed), *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*. Lecture Notes in Computer Science 13008, pp. 182–201. Springer.

van der Rest, C. and Poulsen, C. B. (2024) *GitHub - heft-lang/hefty-equations: Modular reasoning about (elaborations of) higher-order effects — github.com*. `https://github.com/heft-lang/hefty-equations`.

van der Rest, C., Poulsen, C. B., Rouvoet, A., Visser, E. and Mosses, P. D. (2022) Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* **6**(OOPSLA2):1903–1932.

Vezzosi, A., Mörtberg, A. and Abel, A. (2021) Cubical agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.* **31**:e8.

Wadler, P. (1992) The essence of functional programming. Sethi, R. (ed), *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992* pp. 1–14. ACM Press.

Wadler, P., Kokke, W. and Siek, J. G. (2020) *Programming Language Foundations in Agda*.

Wand, M. (1979) Final algebra semantics and data type extensions. *J. Comput. Syst. Sci.* **19**(1):27–44.

Wu, N. and Schrijvers, T. (2015) Fusion for free - efficient algebraic effect handlers. Hinze, R. and Voigtländer, J. (eds), *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Lecture Notes in Computer Science 9129, pp. 302–322. Springer.

Wu, N., Schrijvers, T. and Hinze, R. (2014) Effect handlers in scope. Swierstra, W. (ed), *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014* pp. 1–12. ACM.

Yang, Z. and Wu, N. (2021) Reasoning about effect interaction by fusion. *Proc. ACM Program. Lang.* **5**(ICFP):1–29.

Yang, Z., Paviotti, M., Wu, N., van den Berg, B. and Schrijvers, T. (2022) Structured handling of scoped effects. Sergey, I. (ed), *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 13240, pp. 462–491. Springer.

Zhang, Y. and Myers, A. C. (2019) Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* **3**(POPL):5:1–5:29.