Hefty Algebras

for Higher-Order Algebraic Operations

CASPER BACH POULSEN, Delft University of Technology, Netherlands

Algebraic effects and handlers is an increasingly popular approach to programming with effects. An attraction of the approach is its modularity: effectful programs are written against an interface of effectful operations, and the operational meaning of the operations in the interface can be defined and refined without changing or recompiling programs written against the interface. However, higher-order operations (i.e., operations that have computations as parameters) break this modularity, since higher-order operations cannot in general be declared as interface operations. Instead, they are commonly defined as abbreviations with a fixed operational interpretation. Such abbreviations leak operational details that are supposed to be abstracted away, making it harder to refactor and optimize programs written against the interface. A recent line of research therefore focuses on developing new and improved effect handlers that address this modularity problem with higher-order operations. In this paper, we present a (surprisingly) simple alternative solution to the modularity problem with higher-order operations: we factor the abbreviations commonly used to define higher-order operations into interfaces of their own to fix the abstraction leak. Our solution is as expressive as the existing state of the art in effects and handlers. We present our solution, and compare and contrast our solution with previous approaches, by embedding it and previous approaches in Agda.

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

1 INTRODUCTION

Defining abstractions for programming constructs with side effects is a long standing open problem in programming languages. The goal is to define an interface for (possibly) side effectful operations that encapsulates and hides irrelevant operational details. Such encapsulation makes it easy to refactor, optimize, or even change the behavior of a program, by changing the implementation of the interface. Ideally, importing and composing imported interface implementations should require a minimal amount of glue code.

Algebraic effects and handlers [Plotkin and Pretnar 2009] offers an attractive solution to this problem. The idea is that a programmer defines interfaces of effectful operations that they can program against. Effect handlers then provide separate implementations of each interface. Composing different programs with different interfaces, and applying different handlers to run a program, requires no glue code.

However, algebraic effects are not as expressive as, e.g., monads [Moggi 1989] and monad transformers [Cenciarelli and Moggi 1993; Jaskelioff 2008; Liang et al. 1995]. In particular, so-called *higher-order operations* (i.e., operations that have computations as parameters) such as exception catching or scoping constructs commonly found in monadic programming libraries cannot be defined in terms of algebraic effects and handlers directly. They can, however, be defined as abbreviations of more primitive effects and handlers. But such abbreviations represent abstraction leaks: they specialize higher-order operations to a particular operational interpretation, which makes it harder to, e.g., refactor and optimize programs that use higher-order operations.

This problem was first identified by Wu et al. [2014] who proposed *scoped effects and handlers* [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] as an alternative to algebraic effects and handlers. Scoped effects and handlers have similar modularity benefits as algebraic effects and

Author's address: Casper Bach Poulsen, Delft University of Technology, Netherlands, c.b.poulsen@tudelft.nl.

2020. 2475-1421/2020/9-ART1 $0
https://doi.org/0

handlers, but bring these benefits to these a wider class of programming constructs, including many higher-order operations. However, van den Berg et al. [2021] recently observed that certain control-flow constructs that defer computations, such as evaluation strategies for λ application or *(multi-)staging* [Taha and Sheard 2000], are beyond the expressiveness of scoped effects. Furthermore, scoped effects require glue code for *effect weaving* [Wu et al. 2014], which algebraic effects does not. (We discuss the nature of this glue code in § 2.5 of this paper.)

In this paper, we present an alternative and (surprisingly) simple solution to the modularity problem with higher-order operations and effect handlers, using only off-the-shelf existing techniques known from, e.g., *data types à la carte* [Swierstra 2008]. Our solution embraces the view that higher-order operations abbreviate more primitive effects and handlers. But rather than define such abbreviations in an ad hoc, non-modular manner, we propose to treat higher-order operations as an interface whose implementation can be given by modular elaborations into algebraic effects and handlers. We show that our framework avoids the need for effect weaving glue code, while providing similar modularity benefits and expressiveness as scoped effects. We also show that our solution lets us verify algebraic laws about handlers.

In order to compare and contrast with existing solutions, and to place our solution on a solid foundation, we develop our solution as an embedded domain-specific language in Agda, and our paper is a literate Agda file. While the embedding of our framework in Agda makes use of dependent types, the concepts should be readily encodable in less dependently-typed languages like Haskell, OCaml, or Scala.[1]

We make the following technical contributions:

- § 2.1 describes how to encode algebraic effects in Agda, and discusses the modularity problem with higher-order operations in § 2.1. We also summarize how scoped effects and handlers address the modularity problem, for some but not all higher-order operations.
- § 3 presents a surprisingly simple alternative 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 algebras for folding hefty trees into algebraic effect trees and handlers. These *hefty algebras* motivate the title of this paper.
- § 4 shows that hefty algebras support formal reasoning on a par with algebraic effects and handlers, by verifying algebraic laws of higher-order effects for exception catching.
- § 5 presents examples of how to define higher-order effects from the literature as hefty algebras.

§ 6 discusses related work and § 7 concludes.

2 ALGEBRAIC EFFECTS AND HANDLERS IN AGDA

This section introduces the encoding algebraic effects in Agda that we use throughout the rest of the paper. § 2.1 introduces effects and computation types; § 2.2 introduces the encoding we will use of *effect rows*; and § 2.3 defines effect handlers. Then § 2.4 discusses the problem with defining higher order effects using algebraic effects and handlers, and § 2.5 discusses how scoped effects [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] solves the problem for some but not all higher-order operations (*scoped* operations).

Some familiarity with algebraic effects and handlers is helpful but necessary to read this section. For a gentle introduction to algebraic effects and handlers we refer the reader to the tutorial by Pretnar [2015]. We make some use of dependent types, so a passing familiarity with those is also useful. We do not assume familiarity with Agda, and explain Agda specific notation in footnotes.

[1]Indeed, we developed the first iteration of the framework we present in this paper in Haskell.

The encodings we show in this section are based on previous work or folklore knowledge about how to represent algebraic effects in type theory, except for the encoding of row insertions in § 2.2 which is a variation on existing techniques [Liang et al. 1995; Swierstra 2008].

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*.

In less dependently typed languages like Haskell or Scala, such effect signatures are traditionally [Kammar et al. 2013; Kiselyov and Ishii 2015; Swierstra 2008; Wu et al. 2014] given by a *functor*; i.e., a type of kind Set → Set.[2] However, these functor encodings do not translate well to Agda's type theory. The problem is that Agda the functor based encodings give rise to types that Agda is unable to verify are *strictly positive*[3] [Abbott et al. 2003, 2005] and/or *universe consistent*[4] [Martin-Löf 1984]. Luckily, there is a well-established technique [Abbott et al. 2003, 2005] that we can use to encode effect signatures (and data types in general) differently to avoid these issues. Using that technique, we define the type of effect signatures as a (dependent) record type:[5] [6]

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

We can think of Op as defining a type of effectful operations, and of Ret as a function that tells us the *return type* of each effectful operation.

A key idea of algebraic effects is that programs may contain *multiple different* effects. To this end, we will use the following notion of effect signature union:[7] [8]

```
_⊕_ : Effect → Effect → Effect
Op  (ε₁ ⊕ ε₂) = Op ε₁ ⊎ Op ε₂
Ret (ε₁ ⊕ ε₂) = [ Ret ε₁ , Ret ε₂ ]
```

Two effect signatures are thus unioned by taking the disjoint sum of their operations, and by mapping each operation in the sum to their corresponding return types. We will use signature sums to represent a *row* of different possible effects that computations may use; e.g., $\epsilon_0 \oplus \epsilon_1 \oplus \cdots$.

Now, a computation is given by a tree that defines all possible sequences of effectful operations that can possibly arise during execution. Each operation in such a tree is described by an effect signature. This intuition corresponds to the free monad over an effect signature:

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

[2] Set is the type of types in Agda.

[3] See also Agda's documentation on strict positivity: https://agda.readthedocs.io/en/v2.6.2.2/language/positivity-checking.html

[4] See also Agda's documentation on universe checking: https://agda.readthedocs.io/en/v2.6.2.2/language/universe-levels.html

[5] The **record** keyword defines a *record type* in Agda. The Effect record type has two fields. Record type fields may be dependently typed. For example, the type of the Ret field depends on the Op field type.

[6] 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.

[7] 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.

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

The pure constructor represents a "pure" computation with no side-effects, whereas impure represents a computation whose first step is an operation ($op : \mathsf{Op}\ \epsilon$) whose continuation ($k : \mathsf{Ret}\ \epsilon\ op \rightarrow \mathsf{Free}\ \epsilon\ A$) expects a value of the return type of the operation. It is instructional to look at an example.

Example 2.1. The data type on the left below defines a data type of operations for changing a single cell memory location that stores a natural number. On the right is its corresponding effect signature.

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

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

The effect signature on the right tells us that the put operation does not return any value of interest (⊤ is the unit type), whereas a get operation returns a natural number. Using this effect signature and the free monad, we can write a simple program that increments the current state by one:

```
incr-example : Free State ⊤
incr-example = impure get (λ n → impure (put (n + 1)) pure)
```

The program can be made more readable by using monadic **do** notation, as we show in § 2.2.

The incr-example program above makes use of just a single effect. Say we wanted a program that makes use of another effect, Throw:

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

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

The throw operation represents throwing an exception and aborting a computation. The return type of throw is the empty type, which ensures that the continuation of throw can never be called.

If we want to write a program that uses both State and Throw, we need to make explicit and tedious use of sum injections. For example, the following program which increments the state by one and then raises an exception:[9]

```
incr-throw-example₀ : Free (State ⊕ Throw) A
incr-throw-example₀ =
  impure (inj₁ get) (λ n → impure (inj₁ (put (n + 1))) (λ _ → impure (inj₂ throw) ⊥-elim))
```

To avoid such tedious injections, we will make use of *row insertions* to define *smart constructors* that handle injection automatically.

2.2 Row Insertions, Smart Constructors, and Free Folding

A row insertion $\epsilon \sim \epsilon_0 \blacktriangleright \epsilon'$ is a data type representing a witness that ϵ is the row (i.e., effect signature union) resulting from inserting the effect signature ϵ_0 somewhere in the row ϵ'.

```
data _~_▸_ : Effect → Effect → Effect → Set₁ where
  insert :                 (ε₀ ⊕ ε′) ~ ε₀ ▸ ε′
  sift   : (ε ~ ε₀ ▸ ε′) → ((ε₁ ⊕ ε) ~ ε₀ ▸ (ε₁ ⊕ ε′))
```

The insert constructor witnesses that ϵ_0 is inserted in front of ϵ', whereas sift witnesses that the insertion happens in the tail ϵ' of the row $\epsilon_1 \oplus \epsilon'$. We can instruct Agda to automatically construct

[9] ⊥-elim is the eliminator for the empty type, encoding the *principle of explosion*: $\bot\text{-elim} : \bot \rightarrow A$.

insertion witnesses for us when we need it. To do so, we add the following two functions that abbreviate the constructors in an **instance** scope.[10]

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{insert}_{\blacktriangleright} : (\epsilon_0 \oplus \epsilon') \sim \epsilon_0 \blacktriangleright \epsilon' \\
&\qquad \mathsf{insert}_{\blacktriangleright} = \mathsf{insert} \\
&\qquad \mathsf{sift}_{\blacktriangleright} : \{\!\{\ \epsilon \sim \epsilon_0 \blacktriangleright \epsilon'\ \}\!\} \to ((\epsilon_1 \oplus \epsilon) \sim \epsilon_0 \blacktriangleright (\epsilon_1 \oplus \epsilon')) \\
&\qquad \mathsf{sift}_{\blacktriangleright}\ \{\!\{\ w\ \}\!\} = \mathsf{sift}\ w
\end{aligned}
$$

Instances in Agda behave similarly to type classes in Haskell or implicit arguments in Scala: the Agda type checker will automatically find arguments of the right type to pass to functions that bind *instance parameters*, such as the parameter enclosed in instance argument brackets ($\{\!\{\ \}\!\}$) in the $\mathsf{sift}_{\blacktriangleright}$ function above.

Using insertion witnesses and instance parameters, we can define the following injection functions (implementations elided for brevity) that coerce operations of ϵ_0 and ϵ' to operations of the larger row ϵ:

$$
\begin{aligned}
&\mathsf{inj}_{\blacktriangleright_l} : \{\!\{\ \epsilon \sim \epsilon_0 \blacktriangleright \epsilon'\ \}\!\} \to \mathsf{Op}\ \epsilon_0 \to \mathsf{Op}\ \epsilon \\
&\mathsf{inj}_{\blacktriangleright_r} : \{\!\{\ \epsilon \sim \epsilon_0 \blacktriangleright \epsilon'\ \}\!\} \to \mathsf{Op}\ \epsilon' \to \mathsf{Op}\ \epsilon
\end{aligned}
$$

Using these, we can now define smart constructors for the get and put operations of the state effect. The idea is to construct a single-operation computation that can be sequenced with other computations. Using Agda instance argument search, smart constructors automatically coerce state effect operations into a larger row of effects:

$$
\begin{aligned}
&\text{`}\mathsf{put} : \{\!\{\ \epsilon \sim \mathsf{State} \blacktriangleright \epsilon'\ \}\!\} \to \mathbb{N} \to \mathsf{Free}\ \epsilon\ \top \\
&\text{`}\mathsf{put}\ \{\!\{\ w\ \}\!\}\ n = \mathsf{impure}\ (\mathsf{inj}_{\blacktriangleright_l}\ (\mathsf{put}\ n))\ (\mathsf{pure} \circ \mathsf{proj\text{-}ret}_{\blacktriangleright_l}\ \{\!\{\ w\ \}\!\}) \\
&\text{`}\mathsf{get} : \{\!\{\ \epsilon \sim \mathsf{State} \blacktriangleright \epsilon'\ \}\!\} \to \mathsf{Free}\ \epsilon\ \mathbb{N} \\
&\text{`}\mathsf{get}\ \{\!\{\ w\ \}\!\} = \mathsf{impure}\ (\mathsf{inj}_{\blacktriangleright_l}\ \mathsf{get})\ (\mathsf{pure} \circ \mathsf{proj\text{-}ret}_{\blacktriangleright_l}\ \{\!\{\ w\ \}\!\})
\end{aligned}
$$

These functions use the following type for automatically coercing a larger effect to a smaller effect row:[11]

$$
\mathsf{proj\text{-}ret}_{\blacktriangleright_l} : \{\!\{\ w : \epsilon \sim \epsilon_0 \blacktriangleright \epsilon'\ \}\!\}\ \{op : \mathsf{Op}\ \epsilon_0\} \to \mathsf{Ret}\ \epsilon\ (\mathsf{inj}_{\blacktriangleright_l}\ op) \to \mathsf{Ret}\ \epsilon_0\ op
$$

In order to sequence computations given by smart constructors (and computations in general), we can use the *monadic bind* of the free monad. The bind function is naturally defined in terms of a generic fold over a free monad tree:

$$
\begin{aligned}
&\mathsf{fold} : (A \to B) \to ((op : \mathsf{Op}\ \epsilon)\ (k : \mathsf{Ret}\ \epsilon\ op \to B) \to B) \to \mathsf{Free}\ \epsilon\ A \to B \\
&\mathsf{fold}\ gen\ alg\ (\mathsf{pure}\ x) \qquad = gen\ x \\
&\mathsf{fold}\ gen\ alg\ (\mathsf{impure}\ op\ k) = alg\ op\ (\mathsf{fold}\ gen\ alg \circ k)
\end{aligned}
$$

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

Intuitively, $m \gg= g$ "concatenates" g to all the leaves of the branches of m.

[10] These two instances are *overlapping*, which will cause Agda instance resolution to fail, unless we enable the option `-overlapping-instances`. The rest of the examples in this paper type check in Agda 2.6.2.2 using this option.

[11] The curly braced $\{op : \cdots\} \to$ is an *implicit argument*. When possible, Agda will automatically infer for us what op is when we apply the function, such that we do not have to pass this argument explicitly.

Example 2.2. By implementing a smart constructor for throw ($\text{`throw} : ⦃\, \epsilon \sim \text{Throw} \blacktriangleright \epsilon' \,⦄ \to \text{Free}\ \epsilon\ A$) we can define our example program from before becomes much more readable:

$$\text{incr-throw-example}_1 : ⦃\, \epsilon \sim \text{State} \blacktriangleright \epsilon_1 \,⦄ \to ⦃\, \epsilon \sim \text{Throw} \blacktriangleright \epsilon_2 \,⦄ \to \text{Free}\ \epsilon\ A$$
$$\text{incr-throw-example}_1 = \textbf{do}\ n \leftarrow \text{`get};\ \text{`put}\ (n + 1);\ \text{`throw}$$

The encoding we have given of algebraic effects in this section essentially corresponds to programming against an interface given by a row of effectful operation signatures. In the next section we show how to modularly define implementations of interfaces, using *effect handlers*.

2.3 Effect Handlers

An effect handler is a function that handles an effect occurring in an effect row. For example, the following function handles the state effect:

$$\text{handleSt} : \text{Free}\ (\text{State} \oplus \epsilon)\ A \to \mathbb{N} \to \text{Free}\ \epsilon\ A$$

Here, the $\mathbb{N}$ in the second parameter of handleSt is a *parameter* of the handler, and we say that handleSt is a *parameterized handler*. While Agda will only allow us to apply handleSt if State is the *first* effect signature in an effect row, we can generically move any effect signature occurring in a row to the front, using a row insertion witness:

$$\text{to-front} : ⦃\, w : \epsilon \sim \epsilon_0 \blacktriangleright \epsilon' \,⦄ \to \text{Free}\ \epsilon\ A \to \text{Free}\ (\epsilon_0 \oplus \epsilon')\ A$$

We can define parameterized effect handlers such as handleSt in terms of a generic fold over a Free monad. To this end, we use the following record type where $P : \text{Set}$ is the parameter type of the handler and $G : \text{Set} \to \text{Set}$ is a *return type modifier* for the handler (for example, a handler for Throw would modify the return type of a computation with a Maybe, to indicate that an exception may have abruptly terminated a computation):

$$\textbf{record}\ \text{ParameterizedHandler}\ (\epsilon : \text{Effect})\ (P : \text{Set})\ (G : \text{Set} \to \text{Set}) : \text{Set}_1\ \textbf{where}$$
$$\quad \textbf{field}\ \text{ret}\ : \{A : \text{Set}\} \to A \to P \to G\ A$$
$$\qquad\qquad \text{hdl} : \{A : \text{Set}\}\ (op : \text{Op}\ \epsilon)\ (k : \text{Ret}\ \epsilon\ op \to P \to \text{Free}\ \epsilon'\ (G\ A)) \to P \to \text{Free}\ \epsilon'\ (G\ A)$$

The ret field defines the action that needs to happen when we are done handling an effect and only a pure value remains in the computation being handled. In this case, the effect handler wraps the final value in the return type modifier G. The hdl field defines the action that needs to happen to handle an operation: given an operation, a continuation, and a parameter of type P, the handler can choose to continue the computation by invoking the continuation, or abort the computation by not doing so. The continuation also expects a parameter of type P, since the fold has recursively applied the current parameterized handler in the continuation (i.e., it implements a *deep handler semantics* [Hillerström and Lindley 2018]).

We handle algebraic effects by folding a ParameterizedHandler over a tree as follows:[12]

$$\text{handle} : ⦃\, \epsilon \sim \epsilon_0 \blacktriangleright \epsilon' \,⦄ \to \text{ParameterizedHandler}\ \epsilon_0\ P\ G \to \text{Free}\ \epsilon\ A \to P \to \text{Free}\ \epsilon'\ (G\ A)$$
$$\text{handle}\ h = \text{fold}\ (\lambda\ p \to \text{pure} \circ \text{ret}\ h\ p)\ [\ \text{hdl}\ h\ ,\ (\lambda\ op'\ k' \to \text{impure}\ op' \circ \text{flip}\ k')\ ]$$
$$\qquad\qquad \circ\ \text{to-front}$$

Example 2.3. The handler for State is given by the following record:

$$\text{hSt} : \text{ParameterizedHandler State}\ \mathbb{N}\ \text{id}$$
$$\text{ret hSt}\ x\ \_ = x$$

[12]Here, $\text{flip} : (B \to A \to C) \to A \to B \to C$; $\text{flip}\ f\ x\ y = f\ y\ x$

```
hdl hSt (put m) k n = k tt m
hdl hSt get     k n = k n n
```

Using this handler together with the following effect signature for representing the end of a row

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

and the function end : Free Nil $A \to A$; end (pure x) = x, we can write a simple test for incrementing state:[13]

```
`incr-example : ⦃ ϵ ∼ State ▸ ϵ′ ⦄ → Free ϵ ℕ
`incr-example = do n ← `get; `put (n + 1); `get

incr-test : end (handle hSt `incr-example 0) ≡ 1
incr-test = refl
```

This illustrates how handlers of algebraic effects can be encoded in Agda, to modularly implement the interfaces given by effect signature rows. The implementation is modular in the sense that (1) we can apply a handler via handle hSt to *any* computation that has State effect; and (2) we can change the implementation of the State effect by implementing a different handler without having to change *anything other than the handler itself*. For example, if we want a different state handler that gives us access to the final state, we do not have to modify or recompile `incr-example.

```
hSt₁ : ParameterizedHandler State ℕ (λ X → X × ℕ)

incr-test₁ : end (handle hSt₁ `incr-example 0) ≡ (1 , 1)
incr-test₁ = refl
```

Next we discuss how common *higher-order effects* do not enjoy the same modularity.

2.4 The Modularity Problem with Higher-Order Effects

Say we want to define an effect whose interface is summarized by the CatchM record below, which represents a witness that a given computation type M : Set → Set has at least a higher-order operation catch, and a first-order operation throw:

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

The idea is that throw throws an exception, and catch m_1 m_2 handles any exception thrown during evaluation of m_1 by running m_2 instead. The problem is that we cannot define operations such as catch in terms of effect signatures. The crux of the problem is that effect signatures only lets us declare simple operations that have a continuation; not computations that have *both* computational parameters *and* a continuation.

However, as Plotkin and Pretnar [2009] show, we can encode catch as an abbreviation of more primitive effects and handlers. Concretely, we can define the following handler for Throw effect signature from the end of § 2.2. We define it as a SimpleHandler, whose definition is exactly the same as ParameterizedHandler except it does not propagate any handler parameters:

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

$$\begin{aligned}&\mathsf{hThrow} : \mathsf{SimpleHandler\ Throw\ Maybe}\\ &\mathsf{ret}\ \ \mathsf{hThrow} \qquad\qquad = \mathsf{just}\\ &\mathsf{hdl}\ \mathsf{hThrow}\ \mathsf{throw}\ k = \mathsf{pure\ nothing}\end{aligned}$$

The handler modifies the return type of the computation by decorating it with a Maybe. If no exception is thrown, the ret case simply wraps the yielded value in a just. If an exception is thrown, the handler never invokes the continuation k, thereby aborting the computation and returning nothing instead.

We can encode catch as an abbreviated application of hThrow inside a computation. To do so we will make use of *effect masking* which lets us "weaken" the type of a computation by inserting extra effects in an effect row:

$$\sharp\_ : \{\!\!\{\ \epsilon \sim \epsilon_0 \blacktriangleright \epsilon'\ \}\!\!\} \to \mathsf{Free}\ \epsilon'\ A \to \mathsf{Free}\ \epsilon\ A$$

Now we can encode the following abbreviation which matches the signature of catch, and where handle_0 applies a SimpleHandler:[14]

$$\begin{aligned}&\mathsf{catch} : \{\!\!\{\ w : \epsilon \sim \mathsf{Throw} \blacktriangleright \epsilon'\ \}\!\!\} \to \mathsf{Free}\ \epsilon\ A \to \mathsf{Free}\ \epsilon\ A \to \mathsf{Free}\ \epsilon\ A\\ &\mathsf{catch}\ m_1\ m_2 = (\sharp\ (\mathsf{handle}_0\ \mathsf{hThrow}\ m_1)) \gg\!= (\mathsf{maybe\ pure}\ m_2)\end{aligned}$$

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

The catch abbreviation is simple, and represents a key use case of what effect handlers were designed for [Plotkin and Pretnar 2009]. However, as observed by Wu et al. [2014], programs that use abbreviations 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 abbreviation 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. Thus the catch abbreviation falls short of providing an interface for effectful operations that encapsulates and hides irrelevant details.

The problem affects constructs beyond exception catching. Other examples of operations that we cannot express include the local operation of a reader monad:

$$\begin{aligned}&\mathbf{record}\ \mathsf{ReaderM}\ (R : \mathsf{Set})\ (M : \mathsf{Set} \to \mathsf{Set}) : \mathsf{Set}_1\ \mathbf{where}\\ &\quad \mathbf{field}\ \mathsf{ask}\ \ \ : M\ R\\ &\qquad\quad\ \ \mathsf{local} : (R \to R) \to M\ A \to M\ A\end{aligned}$$

Or even operations representing function abstraction and application as higher-order operations whose handlers decide the evaluation strategy (e.g., call-by-value or call-by-name) [van den Berg et al. 2021]. Even more examples can be found in the literature on scoped effects and handlers [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022]. In the next subsection we describe how to define effectful operations such as the operations summarized by CatchM and ReaderM modularly using scoped effects and handlers, and discuss how this is not possible for, e.g., operations representing λ abstraction.

2.5 Scoped Effects and Handlers

This subsection gives an overview of scoped effects and handlers and their support for higher order effects. 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 show how our solution in § 3 is different.

[14]The maybe function is the eliminator for the Maybe type. The first function defines what to do when given a just; the second case is for nothing. Its type is $\mathsf{maybe} : (A \to B) \to B\ \mathsf{Maybe}\ A \to B$.

Scoped effects extends the expressiveness of algebraic effects to support a class of higher-order operations that Piróg et al. [2018]; Wu et al. [2014]; Yang et al. [2022] dub *scoped operations*. The strengthened expressiveness comes at the cost of requiring interface implementers to provide some additional glue code for *weaving* return type modifications through computations. We illustrate how scoped effects work, using a freer monad encoding of the endo functor algebra approach of Yang et al. [2022]. Note that the work of Yang et al. [2022] does not include any examples of handlers that require weaving. However, weaving is required to define the scoped effects counterpart to the ParameterizedHandler record, as we show below.

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

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

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

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

```
data CatchOp : Set where          Catch : Effect
  catch : CatchOp                 Op  Catch = CatchOp
                                  Ret Catch catch = Bool
```

The effect signature indicates that Catch has two scopes since Bool has two inhabitants. The following declares a smart constructor for Catch:

```
‘catch : ⦃ γ ∼ Catch ▸ γ′ ⦄ → Prog ϵ γ A → Prog ϵ γ A → Prog ϵ γ A
‘catch ⦃ w ⦄ m₁ m₂ = enter (inj▸ₗ catch) (λ b → if (proj-ret▸ₗ ⦃ w ⦄ b) then m₁ else m₂) return
```

Following Yang et al. [2022], we can handle scoped operations using a structure-preserving fold over Prog:

```
hcata :   (∀ {X}   → X                                        → F X)
        → (∀ {X}   → (op : Op ϵ) (k : Ret ϵ op → F X)         → F X)
        → (∀ {B X} → (op : Op γ) (k : Ret γ op → F B) → (B → F X) → F X)
        → Prog ϵ γ A → F A
```

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

We can now define a generic notion of modular handlers akin to the generic ParameterizedHandler type we introduced in § 2.3. The handler for the Catch effect needs to handle an algebraic effect (Throw) and a scoped effect (Catch) *simultaneously*. The following type declares the cases a programmer needs to provide to this end:

```
record SimpleHandlerϵγ (ϵ γ : Effect) (G : Set → Set) : Set₁ where
  field
    ret    :   A → G A
    hcall  :   (op : Op ϵ) (k : Ret ϵ op → Prog ϵ′ γ′ (G X)) → Prog ϵ′ γ′ (G X)
```

$$
\begin{aligned}
\mathsf{henter} : \;& (op : \mathsf{Op}\ \gamma)\ (sc : \mathsf{Ret}\ \gamma\ op \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ B))\ (k : B \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ X)) \\
& \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ X) \\
\mathsf{weave} : \;& (k : C \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ X))\ (r : G\ C) \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ X)
\end{aligned}
$$

The ret and hcall cases are similar to the ret and hdl cases from § 2.3. The henter case allows the handler to invoke scoped sub-computations and inspect their return types, before (optionally) passing control to the continuation k. The weave function is glue code. To see why weave is needed, it is instructional to look at how a SimpleHandlerRS is folded over a Prog:

$$
\begin{aligned}
&\mathsf{handle}\epsilon\gamma_0 : \; ⦃\, w_1 : \epsilon \sim \epsilon_0 \blacktriangleright \epsilon' \,⦄ \to ⦃\, w_2 : \gamma \sim \gamma_0 \blacktriangleright \gamma' \,⦄ \to \mathsf{SimpleHandler}\epsilon\gamma\ \epsilon_0\ \gamma_0\ G \\
&\qquad \to \mathsf{Prog}\ \epsilon\ \gamma\ A \to \mathsf{Prog}\ \epsilon'\ \gamma'\ (G\ A) \\
&\mathsf{handle}\epsilon\gamma_0\ h = \mathsf{hcata} \\
&\quad (\mathsf{return} \circ \mathsf{ret}\ h) \\
&\quad [\ \mathsf{hcall}\ h\ ,\ \mathsf{call}\ ] \\
&\quad [\ \mathsf{henter}\ h \\
&\quad ,\ (\lambda\ op'\ sc'\ k' \to \mathsf{enter}\ op'\ sc'\ (\mathsf{weave}\ h\ k'))\ ] \circ \mathsf{to\text{-}front}\epsilon \circ \mathsf{to\text{-}front}\gamma
\end{aligned}
$$

The last line above shows how weave is used. Because we have eagerly folded 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 actually returns ($G\ B$). The weave function weaves the return type modification G into the continuation.

The handler for exception catching using scoped effects is thus:

$$
\begin{aligned}
&\mathsf{hCatch} : \mathsf{SimpleHandler}\epsilon\gamma\ \mathsf{Throw}\ \mathsf{Catch}\ \mathsf{Maybe} \\
&\mathsf{ret} \quad\ \ \mathsf{hCatch}\ x = \mathsf{just}\ x \\
&\mathsf{hcall} \quad \mathsf{hCatch}\ \mathsf{throw}\ k = \mathsf{return}\ \mathsf{nothing} \\
&\mathsf{henter} \ \ \mathsf{hCatch}\ \mathsf{catch}\ sc\ k = \mathbf{let}\ m_1 = sc\ \mathsf{true};\ m_2 = sc\ \mathsf{false}\ \mathbf{in} \\
&\quad m_1 \gg\!= \mathsf{maybe}\ k\ (m_2 \gg\!= \mathsf{maybe}\ k\ (\mathsf{return}\ \mathsf{nothing})) \\
&\mathsf{weave} \ \ \mathsf{hCatch}\ k = \mathsf{maybe}\ k\ (\mathsf{return}\ \mathsf{nothing})
\end{aligned}
$$

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 weave field of hCatch says that, if an unhandled exception is raised during evaluation of a scope, the continuation is discarded and the exception is propagated; and if no exception is raised, the continuation proceeds normally.

As observed by van den Berg et al. [2021], some higher-order effects do not correspond to scoped operations. In particular, the LambdaM record shown below § 2.4 is not a scoped operation.

$$
\begin{aligned}
&\mathbf{record}\ \mathsf{LambdaM}\ (V : \mathsf{Set})\ (M : \mathsf{Set} \to \mathsf{Set}) : \mathsf{Set}_1\ \mathbf{where} \\
&\quad \mathbf{field}\ \mathsf{lam} : (V \to M\ V) \to M\ V \\
&\qquad\qquad \mathsf{app} : V \to M\ V \quad \to M\ V
\end{aligned}
$$

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 such that it is evaluation strategy agnostic.

The LambdaM operations are not a scoped operation because their control flow does not correspond to the control flow of a scope. In particular, control does *not* flow from the sub-computation of the lam operation directly to the continuation. Instead, the operation delays the running of the sub-tree. An app operation somewhere else in the computation may then invoke this delayed computation. The difference in control flow is subtle but significant: the flow is neither compatible with call (which has no sub-computations) nor with enter (whose continuation is wired to receive data yielded by a local scope rather than a non-local computation, as it needs to happen with app). It is possible to define an abbreviation that elaborates the operations of the LambdaM record into more primitive algebraic and scoped effects, but such abbreviations are as non-modular as the catch abbreviation.

In the next section we present a solution that supports a broader class of higher-order effects than scoped effects, and which does not require weave glue code.

3 HEFTY TREES AND ALGEBRAS

The previous section discussed the modularity problem with higher-order effects, and how scoped effects solves the problem for some but not all higher-order operations. In this section we present a different solution to the modularity problem with higher-order effects which works for higher-order operations beyond scoped operations. As observed in § 2.4, operations such as catch can be defined as abbreviations of more primitive effects and handlers. However, these abbreviations represent leaky abstractions. The solution that we propose to solve this problem is to factor these abbreviations so into interfaces of their own to fix the abstraction leak.

To this end, we first introduce a type of *h*igher-order *ef*fec*t* trees (*hefty* trees in § 3.1) which represents the syntax of effectful programs with higher-order operations. Subsequently, we show how to modularly compose algebras which, when folded over a tree, elaborates a hefty tree into more primitive effects and handlers.

3.1 Hefty Trees

As described in § 2.1, algebraic effect trees are given by the free monad over an effect signature. Higher-order effects have similar effect signatures. But whereas algebraic effect signatures define only the type of effectful operations ($\mathsf{Op} : \mathsf{Set}$) and their return types ($\mathsf{Ret} : \mathsf{Op} \to \mathsf{Set}$), higher-order effect signatures also define the shape of sub-trees for each operation. The shape of a sub-tree is, in turn, given by an effect signature whose Op type defines how many branches the tree has, and whose Ret field defines what the return type is of each branch. The Effect^H type comprises all of this information:

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

Here the Fork field associates with each operation an effect signature which defines the shape of sub-trees. Using this type, higher-order effect trees (or *hefty* trees) is given by the following type:

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

Whereas each impure node of a Free tree (§ 2.1) only branched over the return type of an operation, each impure node in a Hefty tree additionally has Forking branches. These Forking branches represent the computational parameters of higher-order operations. In contrast to enter nodes in scoped effect Programs (§ 2.5), impure nodes in Hefty trees make no assumptions about how control flows. As we show in § 3.2, it is up to algebras to elaborate and stitch together the control-flow of higher-order operations in a well-typed manner.

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

$$
\begin{aligned}
&\mathsf{Lift} : \mathsf{Effect} \to \mathsf{Effect}^H \\
&\mathsf{Op}\ \ (\mathsf{Lift}\ \epsilon) \quad = \mathsf{Op}\ \epsilon \\
&\mathsf{Fork}\ (\mathsf{Lift}\ \epsilon)\ \_ = \mathsf{Nil} \\
&\mathsf{Ret}\ \ (\mathsf{Lift}\ \epsilon) \quad = \mathsf{Ret}\ \epsilon
\end{aligned}
$$

We can also define a similar notion of signature union as in § 2.1:

$$
\begin{aligned}
&\_\dot{+}\_ : \mathsf{Effect}^H \to \mathsf{Effect}^H \to \mathsf{Effect}^H \\
&\mathsf{Op}\ \ (H_1 \dot{+} H_2) = \mathsf{Op}\ H_1 \uplus \mathsf{Op}\ H_2 \\
&\mathsf{Fork}\ (H_1 \dot{+} H_2) = [\ \mathsf{Fork}\ H_1\ , \mathsf{Fork}\ H_2\ ] \\
&\mathsf{Ret}\ \ (H_1 \dot{+} H_2) = [\ \mathsf{Ret}\ H_1\ \ , \mathsf{Ret}\ H_2\ \ ]
\end{aligned}
$$

And a similar notion of row insertion:

$$
\begin{aligned}
&\mathbf{data}\ \_\sim\_\triangleright\_ : \mathsf{Effect}^H \to \mathsf{Effect}^H \to \mathsf{Effect}^H \to \mathsf{Set}_1\ \mathbf{where} \\
&\quad \mathsf{insert} : \qquad\qquad\qquad\quad (H_0 \dot{+} H') \sim H_0 \triangleright H' \\
&\quad \mathsf{sift} \quad : H \sim H_0 \triangleright H' \to (H_1 \dot{+} H) \sim H_0 \triangleright (H_1 \dot{+} H')
\end{aligned}
$$

Using these, we can define smart constructors, such as the following smart constructor for the Lift effect, which lifts any algebraic operation to a higher-order operation:

$$
\uparrow\_ : \{\!\!\{\ w : H \sim \mathsf{Lift}\ \epsilon \triangleright H'\ \}\!\!\} \to (op : \mathsf{Op}\ \epsilon) \to \mathsf{Hefty}\ H\ (\mathsf{Ret}\ \epsilon\ op)
$$

We can also 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:[15]

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{CatchOp}^d : \mathsf{Set}_1\ \mathbf{where} \\
&\quad \mathsf{catch}^d : \mathsf{Set} \to \mathsf{CatchOp}^d
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Catch}^d : \mathsf{Effect}^H \\
&\mathsf{Op}\ \ \ \mathsf{Catch}^d = \mathsf{CatchOp}^d \\
&\mathsf{Fork}\ \mathsf{Catch}^d\ (\mathsf{catch}^d\ A) = \mathbf{record} \\
&\quad \{\ \mathsf{Op} = \mathsf{Bool};\ \mathsf{Ret} = \lambda\ \_ \to A\ \} \\
&\mathsf{Ret}\ \ \mathsf{Catch}^d\ (\mathsf{catch}^d\ A) = A
\end{aligned}
$$

The Fork field of the signature on the right indicates that Catch has two sub-computations (since Bool has two constructors), and that the return type of each sub-computation is the type A which the catch^d operation is parameterized by. There is just one problem: the signature on the right above is now well-typed!

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 into the Effect^H type which requires operations to live in Set. Luckily, this problem has a well-known solution: *universes of types* [Martin-Löf 1984]. A universe of types is a (dependent) pair of a type that defines the syntax of types ($\mathsf{T} : \mathsf{Set}$) and a function that defines the meaning of the syntax by mapping it onto an Agda type ($[\![\_]\!] : \mathsf{Set}$):

$$
\begin{aligned}
&\mathbf{record}\ \mathsf{Universe} : \mathsf{Set}_1\ \mathbf{where} \\
&\quad \mathbf{field}\ \mathsf{T} \quad\ : \mathsf{Set} \\
&\qquad\qquad [\![\_]\!] : \mathsf{T} \to \mathsf{Set}
\end{aligned}
$$

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

Instead of parameterizing our operation by an actual type, we parameterize it by the syntax of a type. Our effect signature then asserts that the return type of each sub-computation of catch, and its overall return type, is the meaning of the syntactic type parameter of catch:

```
data CatchOp (T : Set) : Set where
  catch : T → CatchOp T
```

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

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

3.2 Hefty Algebras

The previous subsection introduced hefty trees as an encoding of effectful programs with higher-order operations. In this subsection we introduce a type of hefty algebras that fold over hefty trees. By encoding elaborations (like the catch abbreviation) in terms of hefty algebras we get a modular framework for defining the syntax and semantics of higher-order operations in a way that does not suffer from the modularity problem discussed in § 2.4.

We define the type of hefty algebras (Alg) as a single-field record:[16]

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

The algebra essentially defines how to fold an impure node of a tree of type Hefty H A into a value of type G A, assuming we have already folded both the forking branches and the continuation branches into G values:

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

We will consider how to define elaborations as algebras shortly. First, we consider how to sequence Hefty trees via monadic binding (_»=_). We cannot use the cata^H function to define the monadic bind directly, because it folds over *both* forks *and* continuations. But to sequence Hefty trees we should *only* fold over continuations. Otherwise, we would concatenate computations to sub-trees that may represent scopes (as in the branches of a catch operation) or delayed computations (as in the body of function abstractions which we will consider as a higher-order effect in § 5.1). The following definition of bind only concatenates the continuation branches of impure nodes:

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

As shown in § 2.4, the catch operation can be encoded as a non-modular abbreviation:

[16]The reason we encode algebras as a single-field record rather than simply a function is that they have a more well-behaved type inference semantics in Agda. In particular, some implicit parameter inferences and instance parameter resolutions may fail (in Agda 2.6.2.2) if we encode algebras as a function instead of a single-field record.

```
catch m₁ m₂ = (♯ (handle₀ hThrow m₁)) »= (maybe pure m₂)
```

We can generalize this abbreviation by defining it in terms of an elaboration from the higher-order operation catch into algebraic effects and handlers instead. Such elaborations have the following type:

```
Elaboration : Effectᴴ → Effect → Set₁
Elaboration H ϵ = Alg H (Free ϵ)
```

An elaboration of this type represents an implementation of a higher-order effect interface by elaborating the higher-order effect into more primitive effects and handlers.

A nice property of elaborations is that they are composable, because algebras are composable.

```
_⋎_ : Alg H₁ F → Alg H₂ F → Alg (H₁ ∔ H₂) F
alg (A₁ ⋎ A₂) (inj₁ op) = alg A₁ op
alg (A₁ ⋎ A₂) (inj₂ op) = alg A₂ op
```

Given a (composed) elaboration, we can generically transform any hefty tree into more primitive algebraic effects and handlers:

```
elaborate : Elaboration H ϵ → Hefty H A → Free ϵ A
elaborate = cataᴴ pure
```

In summary, the only glue code we need to compose implementations of higher-order effects is to compose their elaboration algebras, and to invoke the appropriate effect handlers of the algebraic effects that higher-order operations elaborate to. In § 3.4 we show how Agda can automatically infer elaboration compositions for us. First, we show how to generalize the non-modular abbreviation of catch by defining it as an Elaboration instead:

```
eCatch : ⦃ u : Universe ⦄ ⦃ w : ϵ ∼ Throw ▸ ϵ′ ⦄ → Elaboration Catch ϵ
alg eCatch (catch t) ψ k = let m₁ = ψ true; m₂ = ψ false in
  (♯ (handle₀ hThrow m₁)) »= maybe k (m₂ »= k)
```

The elaboration is essentially the same as its non-modular counterpart, except that it now uses the universe of types encoding discussed in § 3.1, and that it now transforms syntactic representations of higher-order operations instead. The next subsection shows how hefty trees and algebras supports programming against an interface of side effectful, higher-order operations (such as catch) and modularly composing implementations (such as eCatch) of this interface.

3.3 Programming with Hefty Trees and Algebras

The following program uses the smart constructor for the higher-order operation catch described by the Catch signature, and uses Lift to embed algebraic operations into a higher-order effect tree.

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

The program first sets the state to the value 1; then sets it to 2 and raises an exception handled by the enclosing ‵catch operation; and finally gets the final state. Here ⊥-elim is the eliminator for the empty type which is the return type of the lifted throw operation. Furthermore, the program is making use of the following simple universe of types which Agda has automatically resolved for us, using instance argument resolution:

```
data Type : Set where
  unit  : Type
  num : Type
```

```
TypeUniverse : Universe
T     ⦃ TypeUniverse ⦄ = Type
⟦_⟧ ⦃ TypeUniverse ⦄ unit  = ⊤
⟦_⟧ ⦃ TypeUniverse ⦄ num = ℕ
```

To run the transact, we need to elaborate both the Catch operation and the lifted algebraic operations. Lifted algebraic operations are generically elaborated into their corresponding algebraic effects via the following elaboration algebra:

```
eLift : ⦃ ϵ ∼ ϵ₀ ▸ ϵ′ ⦄ → Elaboration (Lift ϵ₀) ϵ
alg (eLift ⦃ w ⦄) op ψ k = impure (inj▸ₗ op) (k ∘ proj-ret▸ₗ ⦃ w ⦄)
```

The following function composes the elaborations for the row of higher-order effects that transact uses.

```
transact-elab : Elaboration (Lift State ∔ Lift Throw ∔ Catch ∔ Lift Nil) (State ⊕ Throw ⊕ Nil)
transact-elab = eLift ⋎ eLift ⋎ eCatch ⋎ eNil
```

We run a program by first elaborating it and then invoking the relevant handlers for State and Throw (§ 2.3 and § 2.4):

```
test-transact : end (handle₀ hThrow (handle hSt (elaborate transact-elab transact) 0))
                ≡ just 2
test-transact = refl
```

We can also modularly change the meaning of Catch by using the following different elaboration which treats exceptions as being non recoverable.

```
eCatch₁ : ⦃ u : Universe ⦄ ⦃ w : ϵ ∼ Throw ▸ ϵ′ ⦄ → Elaboration Catch ϵ
alg eCatch₁ (catch t) ψ k = (ψ true) »= k
```

Now, without making any changes to the original transact program or recompiling any code:

```
transact-elab₁ : Elaboration (Lift State ∔ Lift Throw ∔ Catch ∔ Lift Nil) (State ⊕ Throw ⊕ Nil)
transact-elab₁ = eLift ⋎ eLift ⋎ eCatch₁ ⋎ eNil

test-transact₁ : end (handle₀ hThrow (handle hSt (elaborate transact-elab₁ transact) 0))
                 ≡ nothing
test-transact₁ = refl
```

The examples above show how hefty trees and algebras supports programming against an interface of side effectful, higher-order operations and modularly composing implementations of this interface. Unlike with scoped effects, elaborations do not require programmers to define weaving functions manually. Elaborations are, however, explicitly composed, but these compositions are routine and can be automated using Agda instance arguments.

3.4 Automating Elaboration Composition

A naive approach to automating elaboration compositions is to define a function which automatically resolves elaborations for any effect signature sum:

```
auto-elab₀ : ⦃ E₁ : Elaboration H₁ ϵ ⦄ ⦃ E₂ : Elaboration H₂ ϵ ⦄ → Elaboration (H₁ ∔ H₂) ϵ
auto-elab₀ ⦃ E₁ ⦄ ⦃ E₂ ⦄ = E₁ ⋎ E₂
```

However, instance resolution using this function does not behave well in practice (in Agda 2.6.2.2): signature sums are given by a function which Agda (partially) unfolds for us, and the unfolded definitions are not straightforwardly unifiable by Agda's instance resolution engine. Similarly to how we chose in § 3.2 to represent algebras as a single-field record, we can wrap elaborations in a single-field record, and define the auto-elab function in terms of that record type instead:

```
record Elab (H : Effect^H) (ε : Effect) : Set₁ where
  field orate : Alg H (Free ε)

elab : Elab H ε → Hefty H A → Free ε A
elab = elaborate ∘ orate

instance
  auto-elab : ⦃ E₁ : Elab H₁ ε ⦄ ⦃ E₂ : Elab H₂ ε ⦄ → Elab (H₁ ∔ H₂) ε
  orate (auto-elab ⦃ E₁ ⦄ ⦃ E₂ ⦄) = (orate E₁) ⋎ (orate E₂)
```

Adding eLift and eCatch as Elab instances, we can automatically compose elaborations:

```
transact-elab′ : Elaboration (Lift State ∔ Lift Throw ∔ Catch ∔ Lift Nil) (State ⊕ Throw ⊕ Nil)
transact-elab′ = orate auto-elab
```

We have shown how hefty trees and algebras lets us modularly elaborate an interface of higher-order effects into more primitive effects and handlers. By elaborating to effects and handlers we inherit their expressiveness. In § 5 we show by example how this expressiveness lets us define a wide range of programming language constructs, including constructs that are beyond the expressiveness of, e.g., scoped effects (in particular, λ abstraction in § 5.1).

4 VERIFYING ALGEBRAIC LAWS FOR HIGHER-ORDER EFFECTS

A key idea of algebraic effects is to provide an interface that defines the syntax of operations but also algebraic laws about the operations. In this section we show how to verify the lawfulness of the catch higher-order effect, and compare the effort required to verify lawfulness using hefty algebras vs. a non-modular abbreviation for catch.

The record type shown below defines the interface of a monad (given by the M, *return*, and _»=_ parameters of the record) with a throw and bind operation. The fields on the left below assert that M has a *throw* and *catch* operation, as well as a run function which runs a computation to produce a result R : Set → Set.[17] On the right below are the laws that constrain the behavior of the throw and catch operations. The laws are borrowed from Delaware et al. [2013].

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

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

Below we show that the elaboration and handler from the previous section satisfy these laws. The proofs are equational rewriting proofs akin to the ones you would write on pen-and-paper, except

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

each step is mechanically verified. More interesting than the proofs themselves is the question of how much overhead the hefty algebra encoding adds compared to the non-modular abbreviation of catch from § 2.4. To answer this question show two different implementations of CatchIntf: one that uses hefty algebras ($\mathsf{CatchImpl}_0$ on the left), and one that uses the non-modular abbreviation of catch ($\mathsf{CatchImpl}_1$ on the right). The implementations use $`\mathsf{throw}^H$ as an abbreviation for ↑ throw »= ⊥-elim, h as an abbreviation of the handler for the throw operation, and e as an abbreviation of the elaboration function. Each proof consists of a series of equational rewrites (using the ≡-Reasoning module from the Agda standard library) of the form $t_1 \equiv\langle\ eq\ \rangle\ t_2$ where t_1 is the term before the rewrite, t_2 is the term after, and eq is a proof that t_1 and t_2 are equal.

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

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

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

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

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

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

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

5 EXAMPLES

As discussed in § 2.4, there is a wide range of examples of higher-order effects that cannot be defined as algebraic operations directly, but only in terms of abbreviations and leaky abstractions. In this section we give examples of such effects and show to define them modularly using hefty algebras. The code accompanying the paper contains the full examples.

5.1 λ as a Higher-Order Operation

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

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

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

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

Here ‵lam constructs a higher-order operation with a sub-tree parametric in a thunk value of type $⟦\ \mathsf{c}\ t_1\ ⟧$. The return type of the higher-order operation is a function value of type $⟦\ \mathsf{c}\ t_1 \rightarrowtail t_2\ ⟧$ where the parameter type matches the type of the function body. The ‵var operation accepts a thunk value as argument and yields a value type that matches the thunk 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* whose return type matches the function value parameter type.

The interface above defines a kind of *higher-order abstract syntax* [Pfenning and Elliott 1988] which piggy-backs on Agda functions for name binding. However, unlike most Agda functions, the constructors above represent functions with side-effects. The representation in principle supports functions with arbitrary side-effects since it is parametric in what the higher-order effect signature H is. Furthermore, we can assign different operational interpretations to the operations in the interface without having to change the interface or programs written against the interface. To illustrate we give two different implementations of this interface: one that implements a call-by-value evaluation strategy, and one that implements call-by-name.

5.1.1 Call-by-Value. We give a call-by-value interpretation of the higher-order operations for λ by elaborating to algebraic effect trees with effects ϵ. 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 : T} → ⟦ t₁ ↣ t₂ ⟧ ↔ (⟦ t₁ ⟧ → Free ε ⟦ t₂ ⟧)
iso₂ : {t : T}     → ⟦ c t     ⟧ ↔ ⟦ t ⟧
```

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

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

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

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

```
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 : Elab (Lam ∔ Lift State ∔ Lift Nil) (State ⊕ Nil)
elab-cbv = auto-elab

test-ex-cbv : end (handle hSt (elab elab-cbv ex) 0) ≡ 4
test-ex-cbv = refl
```

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

$$\begin{aligned}
&\mathsf{iso}_1 : \{t_1\ t_2 : \mathsf{T}\} \to [\![\ t_1 \rightarrowtail t_2\ ]\!] \leftrightarrow ([\![\ t_1\ ]\!] \to \mathsf{Free}\ \epsilon\ [\![\ t_2\ ]\!])\\
&\mathsf{iso}_2 : \{t : \mathsf{T}\} \quad\ \ \to [\![\ \mathsf{c}\ t\ ]\!] \qquad\ \leftrightarrow \mathsf{Free}\ \epsilon\ [\![\ t\ ]\!]
\end{aligned}$$

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

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

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

```
elab-cbn : Elab (Lam ∔ Lift State ∔ Lift Nil) (State ⊕ Nil)
elab-cbn = auto-elab

test-ex-cbn : end (handle hSt (elab elab-cbn ex) 0) ≡ 5
test-ex-cbn = refl
```

5.2 Optionally Transactional Exception Catching

A feature of scoped effect handlers [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] is that changing the order of handlers makes it possible to obtain different semantics of *effect interaction*. A classical example of effect interaction is the interaction between state and exception catching. The transact program from § 3.3 illustrates the interaction:

```
transact : ⦃ w_s : H ∼ Lift State ▹ H′ ⦄ ⦃ w_t : H ∼ Lift Throw ▹ H″ ⦄ ⦃ w : H ∼ Catch ▹ H″ ⦄
          → Hefty H ℕ
transact = do
  ↑ put 1
  ‘catch (do ↑ put 2; ‘throw^H) (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 abruptly terminates (throws an exception).

With monad transformers [Cenciarelli and Moggi 1993; Liang et al. 1995] we can recover either of these semantics by permuting the order of monad transformers. With scoped effect handlers we can also recover either by permuting the order of handlers. However, the eCatch elaboration in § 3.2 always gives us a global interpretation of state. In this section we demonstrate how we can recover a transactional interpretation of state by using a different elaboration of the catch operation into an algebraically effectful program with the throw operation and the off-the-shelf *sub/jump* control effects due to Fiore and Staton [2014]; Thielecke [1997]. The different elaboration is modular in the sense that we do not have to change the interface of the catch operation nor any programs written against the interface.

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

$$
\begin{aligned}
&\text{`sub} \;\; : \{t : \mathsf{T}\} \to (Ref\ t \to \mathsf{Free}\ \epsilon\ A) \to (\llbracket\ t\ \rrbracket \to \mathsf{Free}\ \epsilon\ A) \to \mathsf{Free}\ \epsilon\ A \\
&\text{`jump} : \{t : \mathsf{T}\} \to Ref\ t \to \llbracket\ t\ \rrbracket \qquad\qquad\qquad\qquad\qquad\quad \to \mathsf{Free}\ \epsilon\ B
\end{aligned}
$$

An operation `sub $f$ $g$ gives the computation in $f$ access to the continuation $g$ via a reference value $Ref\ t$ which represents a continuation that expects a value of type $\llbracket\ t\ \rrbracket$. The `jump operation lets us invoke such continuations. The two operations and their handler (abbreviated to h) satisfy the following laws (among others):

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

The last of these laws assert that `sub and `jump are, indeed, *algebraic* operations, since their computational sub-terms behave as continuations. Consequently, we encode `sub and its handler as an algebraic effect.

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

```
eCatchCC : Elaboration Catch ϵ
alg eCatchCC (catch x) ψ k = let m₁ = ψ true; m₂ = ψ false in
  `sub
    (λ r → (♯ (handle₀ hThrow m₁)) »= 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. If an exception is raised, then 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 become pre-applied to a state handler that only knows about the “old” state. This is exactly what happens when we invoke the state handler before the handler for sub/jump: in this case we get the transactional interpretation of state, such that running transact program gives 1. Otherwise, if we run the sub/jump handler before the state handler, we get the global interpretation of state, and the result 2.

The sub/jump elaboration above is more involved than the scoped effect handler that we considered in § 2.5. However, the complicated encoding does not pollute the higher-order effect interface, and only turns up if we strictly want or need effect interaction.

5.3 Logic Programming

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

$$\begin{aligned}
&_\text{`or}^{H}_ : \{\!\!\{\, H \sim \text{Lift Choice} \triangleright H' \,\}\!\!\} \to \text{Hefty}\ H\ A \to \text{Hefty}\ H\ A \to \text{Hefty}\ H\ A \\
&\text{`fail}^{H} \quad : \{\!\!\{\, H \sim \text{Lift Choice} \triangleright H' \,\}\!\!\} \qquad\qquad\qquad\qquad\quad\ \to \text{Hefty}\ H\ A
\end{aligned}$$

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

$$\text{`once} : \{\!\!\{\, w : H \sim \text{Once} \triangleright H' \,\}\!\!\}\ \{t : \mathsf{T}\} \to \text{Hefty}\ H\ [\![\ t\ ]\!] \to \text{Hefty}\ H\ [\![\ t\ ]\!]$$

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

```
eOnce : ⦃ ε ~ Choice ▸ ε′ ⦄ → Elaboration Once ε
alg eOnce once ψ k = do
  l ← ♯ (handle₀ hChoice (ψ tt))
  maybe k `fail (head l)
```

The elaboration runs the branch (ψ) of the once operation under the hChoice handler which computes a list of all possible solutions for the sub-tree. Then we try to choose the first solution and pass that to the continuation k. If the sub-tree does not have any solutions, we fail. Under a strict evaluation order, the elaboration above would compute all possible solutions which is doing more work than needed. Agda 2.6.2.2 does not have a specified evaluation strategy, but does compile to Haskell which is lazy. In Haskell, the solutions would be lazily computed, such that the once operator cuts search short as intended.

5.4 Concurrency

The final example we consider are higher-order operations for concurrency, inspired by the *resumption monad* [Claessen 1999; Piróg and Gibbons 2014; Schmidt 1986]. We summarize our encoding and then discuss the relationship with resumption monad.

Our goal is to define two higher-order operations:

$$\begin{aligned}
&\text{`spawn}\ \ : \{t : \mathsf{T}\} \to (m_1\ m_2 : \text{Hefty}\ H\ [\![\ t\ ]\!]) \to \text{Hefty}\ H\ [\![\ t\ ]\!] \\
&\text{`atomic} : \{t : \mathsf{T}\} \to \text{Hefty}\ H\ [\![\ t\ ]\!] \qquad\qquad\quad \to \text{Hefty}\ H\ [\![\ t\ ]\!]
\end{aligned}$$

The operation `spawn $m_1$ $m_2$ spawns two threads that run concurrently, and returns the value produced by $m_1$ once both threads have finished running. The operation `atomic m represents an atomic block to be executed atomically (without the other thread performing any side-effects meanwhile).

We elaborate `spawn by interleaving the sub-trees of its computations. To this end, we use a dedicated function which takes two trees as input, and returns the tree resulting from interleaving the operations in the two trees. This dedicated function should, however, not interleave code that resides in atomic blocks. The following function implements this functionality:

$$\begin{aligned}
\text{interleave}_l :\ &\{Ref : \mathsf{T} \to \text{Set}\} \to \text{Free}\ (\text{ABlock}\ Ref \oplus \epsilon)\ A \to \text{Free}\ (\text{ABlock}\ Ref \oplus \epsilon)\ B \\
&\to \text{Free}\ (\text{ABlock}\ Ref \oplus \epsilon)\ A
\end{aligned}$$

The function takes two trees with atomic blocks as input, and produces a tree with atomic blocks as output. The output tree contains the actions of the two trees in interleaved order, and returns as final result the value yielded by the left input tree.

The ABlock effect that interleave_l uses is the same effect as the CC effect we used in § 5.2.2. We use it to explicitly delimit blocks that should not be interleaved. In this sense, it corresponds to what Wu et al. [2014, § 7] call *scoped syntax*.

Using the interleave_l function and the ABlock effect, the concurrency primitives spawn and atomic are elaborated as follows:

```
eConcur : {Ref_a : T → Set} ⦃ w_a : ε ∼ ABlock Ref_a ▸ ε'' ⦄ → Elaboration Concur ε
alg eConcur (spawn t) ψ k =
  from-front (interleave_l (to-front (ψ true)) (to-front (ψ false))) »= k
alg eConcur (atomic t) ψ k = ‵a-block (ψ tt) »= k
```

The from-front and to-front functions use the row insertion witness w_a to move the ABlock effect to the front of the row and back again. To observe the interleaving evaluation order we will make use of the following algebraic effect for printing output:

```
‵out : ⦃ ε ∼ Out X ▸ ε' ⦄ → X → Free ε ⊤

hOut : SimpleHandler (Out X) (_× List X)
ret  hOut x          = x , []
hdl hOut (out x) k = do (v , xs) ← k tt; pure (v , x :: xs)
```

The example program below uses the Out and the Concur effects to spawn two threads. The first thread prints 0, 1, and 2; the second prints 3 and 4.

```
ex-01234 : Hefty (Lift (Out ℕ) ∔ 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 : end (handle₀ hOut (handle₀! hABlock (elab concur-elab ex-01234)))
               ≡ (0 , 0 :: 3 :: 1 :: 4 :: 2 :: [])
test-ex-01234 = refl
```

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

```
ex-01234567 : Hefty (Lift (Out ℕ) ∔ 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 : end (handle₀ hOut (handle₀! hABlock (elab concur-elab ex-01234567)))
                  ≡ (0 , 0 :: 5 :: 6 :: 7 :: 3 :: 1 :: 4 :: 2 :: [])
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]. The scoped effects definition of [Yang et al. 2022] does not (explicitly) consider how to define the concurrency operations in a modular style. Instead, the scoped effects definition gives 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):

$$\begin{array}{l}\textbf{data}\ \mathsf{Resumption}\ \epsilon\ A : \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{done} : A \qquad\qquad\qquad\qquad\quad \rightarrow \mathsf{Resumption}\ \epsilon\ A\\ \quad \mathsf{more} : \mathsf{Free}\ \epsilon\ (\mathsf{Resumption}\ \epsilon\ A) \rightarrow \mathsf{Resumption}\ \epsilon\ A\end{array}$$

However, elaborating into this resumption monad via a hefty algebra of type Alg Conc (Resumption ϵ) is incompatible with our other elaborations which elaborate into the free monad. For that reason, the elaboration we gave above does not use the resumption monad directly, but emulates it using the free monad.

6 RELATED WORK

As stated in the introduction of this paper, defining abstractions for programming constructs with side effects is a long standing open problem in programming languages. It is, however, also a problem that has received considerable attention over the years. Notably, Moggi [1989] introduced monads for modeling side effects and structuring programs with side effects. An idea which Wadler [Wadler 1992] helped popularize.

However, monads do not naturally compose. A range of different solutions have been developed to address this issue [Cenciarelli and Moggi 1993; Filinski 1999; Jones and Duponcheel 1993; Jr. 1994]. Of these solutions, monad transformers [Cenciarelli and Moggi 1993; Jaskelioff 2008; Liang et al. 1995] is the more widely adopted solution.

More recently, algebraic effects [Plotkin and Power 2002] was proposed as an alternative to monad transformers which provides a more structured approach to defining effects. This structured approach offers some modularity benefits over monads and monad transformers. In particular, monads and monad transformers "leak" information about the monad. In contrast, operations in algebraic effects represent non-leaky interfaces. Furthermore, monad transformers commonly require glue code to "lift" operations between different layers of monad transformer stacks. While 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 the *exception catching* effect that we also discussed throughout this paper, did not fit into the framework of algebraic effects. *Effect handlers* [Plotkin and Pretnar 2009] were introduced to address this problem. Algebraic effects and handlers has since then been gaining traction as a framework for modeling side effects 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] or the freer monad [Kiselyov and Ishii 2015], but also standalone languages such as Eff [Bauer and Pretnar 2015], Koka [Leijen 2017], Frank [Lindley et al. 2017], and Effekt [Brachthäuser et al. 2020].[18]

As discussed in § 2.4, some modularity benefits of algebraic effects and handlers do not carry over to higher-order effects. Scoped effects and handlers [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] address this shortcoming for *scoped operations*. § 2.5 gave a detailed account of scoped effects.

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

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

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 could 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 provide some support for defining higher-order effects. It is, however, unclear what the denotational semantics is of this feature of these languages. An interesting question for future work is whether the modular elaborations we introduce in this paper could provide a denotational model for this feature.

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.

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 reduction rules that implicitly propagate an open-ended set of *auxiliary entities*. Mosses shows that these auxiliary entities can be used to encode common classes of effects, such as effects that read or emit data, effects that have state, and even control effects [Sculthorpe et al. 2015]. The K Framework [Rosu and Serbanuta 2010] takes a different approach but provides many of the same benefits. These frameworks do not encapsulate operational details but instead make it notationally convenient to program (or specify semantics) with side-effects.

7 CONCLUSION

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

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

ACKNOWLEDGMENTS

REFERENCES

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

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

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

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

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

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

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

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

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

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

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.), Vol. 11275. Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

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

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

Guy L. Steele Jr. 1994. Building Interpreters by Composing Monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 472–492. https://doi.org/10.1145/174675.178068

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

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

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

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

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

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

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

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

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

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

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

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 809–818. https://doi.org/10.1145/3209108.3209166

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

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

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

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

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

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

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 98–113. https://doi.org/10.1145/3331545.3342595

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

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

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

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

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

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

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

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