Monad Transformers and Modular Algebraic Effects

What Binds Them Together

Tom Schrijvers
KU Leuven
Belgium

Maciej Piróg
University of Wrocław
Poland

Nicolas Wu
Imperial College London
United Kingdom

Mauro Jaskelioff
CIFASIS-CONICET
Universidad Nacional de Rosario
Argentina

Abstract

For over two decades, monad transformers have been the main *modular* approach for expressing purely functional side-effects in Haskell. Yet, in recent years algebraic effects have emerged as an alternative whose popularity is growing.

While the two approaches have been well-studied, there is still confusion about their relative merits and expressiveness, especially when it comes to their comparative modularity. This paper clarifies the connection between the two approaches—some of which is folklore—and spells out consequences that we believe should be better known.

We characterise a class of algebraic effects that is modular, and show how these correspond to a specific class of monad transformers. In particular, we show that our modular algebraic effects gives rise to monad transformers. Moreover, every monad transformer for algebraic operations gives rise to a modular effect handler.

CCS Concepts • **Software and its engineering → Functional languages**; *Control structures*; *Coroutines*; • **Theory of computation** → *Categorical semantics*.

Keywords Handlers, Effects, Monads, Transformers

ACM Reference Format:
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 Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3331545.3342595

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342595

1 Introduction

For decades monads [29, 42] have dominated the scene of pure functional programming with effects, and the recent popularisation of algebraic effects & handlers [3, 7, 21, 23, 35] promises to change the landscape. However, with rapid change also comes confusion, and practitioners have been left uncertain about the advantages and limitations of the two competing approaches. This paper aims at clarifying the essential differences and similarities between them.

Working with combinations of multiple different effects demands a *modular* approach, where each effect is given semantics separately. This allows for the construction of complex custom effects from off-the-shelf building blocks.

A popular approach to achieving modularity for monads is with monad transformers [27]. Monad transformers extend an arbitrary monad with a new effect while at the same time ensuring that original effects are available. The desired combination of monads is achieved by stacking several monad transformers in a particular order.

In the algebraic effects approach modularity is conceptually achieved in two stages. First, the syntax of all operations involved in the effect are defined. Then a program is incrementally interpreted by several handlers, which in turn give the syntax of different effects a semantics.

Since each handler only knows about the part of the syntax of the effect it is handling, a modular approach to algebraic effects must provide a way of leaving unknown syntax uninterpreted and to be dealt with later by other handlers.

There are several properties that can be studied when comparing approaches, such as expressivity, ease of use, modularity, boilerplate automation, and efficiency. This paper focuses only on the essential *expressivity* of transformers and algebraic effects, leaving other important properties out of scope. To study expressivity, we formulate a minimal implementation of both approaches and abstract over all other aspects. In this way, we aim to explain the main ideas in an approachable manner and provide general insights that can be applied to everyone's favourite Haskell library. Much of what we present is folklore among experts, but the tale has yet to be collected in a single, consistent place for the wider Haskell community to enjoy, as we have done here.

After an introduction to monad transformers (Section 2) that fixes notation, the contributions of this paper are:

1. a novel characterisation of *modular handlers* as type constructors that form a parametric family of Eilenberg-Moore algebras (Section 3). This captures the essence of a class of handlers that are completely independent of one another.
2. a comparison of expressiveness of monad transformers and modular algebraic effects (Section 4), showing that:
   a. every algebraic effect signature gives rise to a monad subclass, and every associated modular effect handler gives rise to a monad transformer that instantiates the class, (Section 5) and
   b. every monad subclass with only algebraic operations gives rise to an algebraic effect signature *and* every monad transformer instantiating the class gives rise to a modular effect handler for the signature (Section 6).

   These transformations are semantics-preserving.
3. a demonstration that *callCC* can be reformulated in terms of algebraic operations (Section 7).

Finally, related work is discussed (Section 8) before conclusions are drawn (Section 9).

2 Monads and Monad Transformers

This section fixes the notation and laws of functors, monads, and monad transformers, which should be familiar concepts.

2.1 Functors and Monads

A functor is a type f equipped with a function $fmap :: (a \to b) \to f\ a \to f\ b$, subject to two laws:

$$fmap\ id = id \qquad fmap\ g \circ fmap\ f = fmap\ (g \circ f)$$

These laws capture the notion that the contents of a container can be modified without affecting its shape.

A monad is a type m that is a functor equipped with two functions $return :: a \to m\ a$ and $join :: m\ (m\ a) \to m\ a$, that are subject to three laws:

$$join \circ return = id \qquad join \circ fmap\ return = id$$
$$join \circ join = join \circ fmap\ join$$

Intuitively, *return* puts a value into a monadic context, and *join* collapses a nested monadic context. The first two laws state that nesting a context with *return* followed by collapsing with *join* changes nothing, and the third law ensures multiple nested contexts can be collapsed in any order.

The *bind* function, $(\gg=)$, is an alternative to *join*, where:

$$mx \gg= f = join\ (fmap\ f\ mx) \text{ and } join\ mmx = mmx \gg= id$$

The meaning of $mx \gg= f$ is to feed the result(s) of mx to f. This can be implemented by applying f within the context mx, and then collapsing the nested context with *join*.

Functors and monads are defined using two type classes (Figure 1), whose instances must respect the laws. For instance, mutable state can be encapsulated by the following:

newtype *State s a* = *State* { *runState* :: $s \to (a, s)$ }

instance *Monad* (*State s*) **where**

return x = *State* ($\lambda s \to (x, s)$)
State p $\gg=$ *k* = *State* ($\lambda s \to$ **let** $(x, s') = p\ s$ **in** *runState* (*k x*) *s'*)

This monad threads a state of type s around, where it can be accessed using *get* or replaced using *put*. The *get* operation returns the state that it is given, leaving the state unchanged, while *put s* changes the state to s.

Instead of relying on their specific implementation as a specification it is better to first consider the properties that these operations satisfy [12]. For instance, state requires:

$$get \gg= put = id \qquad get \gg get = get$$
$$put\ s \gg put\ s' = put\ s' \qquad put\ s \gg get = put\ s \gg return\ s$$

Yet, such operation-specific laws are not the focus of this work; we focus here only on the operations themselves. These operations are then incorporated into a typeclass that embodies stateful computations.

class *Monad m* $\Rightarrow$ *MonadState s m* | $m \to s$ **where**
  get :: *m s*
  put :: $s \to m\ ()$

instance *MonadState s* (*State s*) **where**
  get = *State* ($\lambda s \to (s, s)$)
  put s' = *State* ($\lambda\_ \to ((), s')$)

Monads often come with *run* functions that extract values from monadic computations. One example is the function *runState* :: *State s a* $\to s \to (s, a)$ defined above.

2.2 Monad Transformers

Monad transformers allow monads to be extended with additional functionality by lifting one into another.

Lifting The *Trans* type class is the well-known Haskell interface for monad transformers (Figure 1). This interface provides the monad homomorphism *lift*:

$$lift \circ return = return \qquad lift \circ join = join \circ lift \circ fmap\ lift$$

The laws state that lifting preserves the structure of *return* and *join* from one monad to another.

As an example, the transformer $State_T$ adds *State*-like functionality to an underlying monad.

newtype $State_T\ s\ m\ a = State_T$ { $runState_T :: s \to m\ (a, s)$ }

instance *Trans* ($State_T\ s$) **where**
  lift m = $State_T$ ($\lambda s \to m \gg= \lambda a \to return\ (a, s)$)

instance *Monad m* $\Rightarrow$ *Monad* ($State_T\ s\ m$) **where**
  return x = $State_T$ ($\lambda s \to return\ (x, s)$)
  $State_T\ p \gg= k = State_T$ ($\lambda s \to$ **do** $(x, s') \leftarrow p\ s$
    $runState_T\ (k\ x)\ s'$)

This mirrors its counterpart for *State*, except that the monadic effects of m are threaded through the computation.

```
class Functor f where
  fmap :: (a → b) → f a → f b

class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b

class Trans t where
  lift :: Monad m ⇒ m a → t m a
```

Figure 1. Classes for functors, monads, and transformers

Operations The *Trans* type class only partly provides a transformer's interface: the transformed monad must supply the original operations of interest. In the case of $State_T\ s$, this corresponds to the functions *get* and *put* from *MonadState s*:

```
instance Monad m ⇒ MonadState s (State_T s m) where
  get    = State_T (λs → return (s, s))
  put s' = State_T (λ_ → return ((), s'))
```

Thus, programs like *incr* can be written to work in different contexts, such as *State s* and $State_T\ s\ m$ for any monad *m*:

```
incr :: MonadState Int m ⇒ m ()
incr = get >>= put ∘ succ
```

This small program gets an integer value from the state, and puts back a value that is its successor.

More generally, the operations corresponding to an effect *X* are captured in a monad subclass *MonadX*.

```
class Monad m ⇒ MonadX m where
  op_1 :: ... → m T_1   ;   ...   ;   op_n :: ··· → m T_n
```

As usual, there may be laws associated with this class but we do not focus on them in this paper.

Composition Monad transformers compose by embedding one transformer into another. Given transformers T_1 and T_2 with associated effects given by the classes $MonadX_1$ and $MonadX_2$, then for any monad *m*, either $T_1\ (T_2\ m)$ or $T_2\ (T_1\ m)$ can be produced depending on the order the effects should be interpreted.

It is convenient to use the identity monad *Id* as the monad at the bottom of the stack of transformers. It allows us, for instance, to recover *State s* as $State_T\ s\ Id$.

```
newtype Id a = Id { runId :: a }

instance Monad Id where
  return = Id
  Id x >>= f = f x
```

Here the bind operation is essentially function application.

Instead of directly composing transformers, a specification of class constraints is a cleaner alternative: a computation with the type $(MonadX_1\ m, MonadX_2\ m) ⇒ m\ a$ could be satisfied by either $T_1\ (T_2\ Id)$, or $T_2\ (T_1\ Id)$, thus allowing a fragment of code to be interpreted differently.

For instance, consider the *MonadFail* class, which captures the notion of failing computations.

```
class Monad m ⇒ MonadFail m where
  fail :: m a
```

Only one law is intended to hold, which expresses that no computation is performed after failure:

$$fail >>= f \quad = \quad fail$$

The familiar *Maybe* type has a lawful instance:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x  >>= f = f x

instance MonadFail Maybe where
  fail = Nothing
```

This is well-known to be a monad, where *Just* provides the backbone for successful computations. The operation for *fail* is provided by *Nothing*, which is a left zero of (>>=).

The corresponding transformer is given by $Maybe_T$, where failures are pushed into an underlying monad by wrapping pure values with *Maybe*.

```
newtype Maybe_T m a =
  Maybe_T { runMaybe_T :: m (Maybe a) }

instance Monad m ⇒ Monad (Maybe_T m) where
  return x          = Maybe_T (return (Just x))
  Maybe_T mmx >>= f = Maybe_T
    (do mx ← mmx; case mx of
                    Nothing → return Nothing
                    Just x  → runMaybe_T (f x))

instance Monad m ⇒ MonadFail (Maybe_T m) where
  fail = Maybe_T (return Nothing)

instance Trans Maybe_T where
  lift mx = Maybe_T (fmap Just mx)
```

To lift a computation, we add *Just* to successful results within the monad, and return *Nothing* in the case of failure.

Having defined both *MonadState* and *MonadFail*, Computations that make use of the two effects are easy to express:

```
prog :: (MonadFail m, MonadState Int m) ⇒ m ()
prog = incr >> fail >> incr
```

By keeping the type of this computation abstract, the semantics can be chosen at the point of application. Thus, *prog* can be evaluated with type $Int → Maybe\ ((), Int)$ to return the state only when there are no exceptions, and *Nothing* otherwise. This is achieved by showing how a monad *m* with failure can be promoted through a $State_T$ transformer:

```
instance MonadFail m ⇒ MonadFail (State_T s m) where
  fail = lift fail
```

With this machinery in place, *prog* can be executed with its type specialised to $State_T\ Int\ (Maybe_T\ Id)\ ()$.

```
> (runId ∘ runMaybe_T ∘ flip runState_T 0) prog
Nothing
```

Here the only information that is returned is that *prog* failed.

If we are interested in knowing what the state is even when an error occurs, then we can change the type of the program. Our goal is to get a result of type $(Maybe\ (), Int)$. To achieve this, we can specialise our program to the type $Maybe_T\ (State_T\ Int\ Id)\ ()$, and in order to do so, we must show how a stateful monad can be lifted through $Maybe_T$.

```
instance MonadState s m ⇒
  MonadState s (Maybe_T m) where
    get = lift get
    put = lift ∘ put
```

Now we get a different result:

```
> (runId ∘ flip runState_T 0 ∘ runMaybe_T) prog
(Nothing, 1)
```

This gives back the state just before the *fail* occurred.

Notice that the final computation is the result of running the various transformers one after the other, each interpreting another layer of effects. These functions are an essential part of the interpretation, and often have the general form $run_T :: Monad\ m \Rightarrow T\ m\ A \to m\ B$.

The instances above show a characteristic of the monad transformer approach: the subclass of operations of the underlying monad has to be *lifted* to the transformed monad. For certain transformers, there is a canonical way of lifting operations [16, 18], but in general there might be many different ways to do this, and the choice is dependent on the expected semantics. Therefore, a different instance is usually provided for each combination of monad transformers.

3 Modular Algebraic Effects

3.1 Algebraic Effects

Algebraic Operations The algebraic effects approach restricts itself to so-called *algebraic operations* as the primitive building blocks of monadic computations. Plotkin and Power [33] essentially define algebraic operations for a monad M to be functions *op* of the form:

$$op :: \forall a.\ (M\ a, \ldots, M\ a) \to M\ a$$

and to satisfy the *algebraicity* property

$$op\ (p_1, \ldots, p_n) \gg\!= k = op\ (p_1 \gg\!= k, \ldots, p_n \gg\!= kn) \qquad (1)$$

Conceptually the parameters p_i of an algebraic operation are possible continuations that the operation chooses among.

A Boolean state is modelled by 3 algebraic operations:

$$putT, putF :: \forall a.\ M\ a \to M\ a \quad getB :: \forall a.\ (M\ a, M\ a) \to M\ a$$

The *putT* and *putF* operations overwrite the implicit state with *True* and *False* respectively; they only have one continuation and thus no real choice on how to proceed. The *getB* operation consults the implicit state and chooses the first continuation if it is *True* and the second if it is *False*.

In practice, we typically use equivalent, but more convenient type signatures for algebraic operations. Firstly, we may bundle related primitive operations with the same number of continuation parameters into one combined operation that has an additional parameter to identify the desired primitive operation. For instance, we bundle *putT* and *putF* into

$$putB :: \forall a.\ Bool \to M\ a \to M\ a$$

where the *Bool* parameter allows us to choose between *putT* and *putF*. Secondly, we represent an n-tuple of continuation parameters with a function. For instance, for *getB* we use

$$getB' :: \forall a.\ (Bool \to M\ a) \to M\ a$$

Combining these two variations, the general form for algebraic operations we use is:

$$op :: \forall a.\ A \to (B \to M\ a) \to M\ a$$

where *parameter* type A selects among a number of different primitive operations and *arity* type B indicates the number of possible continuations. More generically, we can say that an algebraic operation's signature has to have the form:

$$op :: \forall a.\ Sig\ A\ B\ (M\ a) \to M\ a$$

where *Sig* is a signature functor for the operation:

```
data Sig a b k = Op a (b → k)
instance Functor (Sig a b) where
  fmap f (Op i k) = Op i (f ∘ k)
```

This yields a nicely pointfree formulation of algebraicity:

$$join \circ op = op \circ fmap\ join \qquad (2)$$

In the following, we define a concrete signature for each effect using functors like *Sig*. In this way, a proper name is given to each operation constructor and isomorphisms can be used to simplify types: we can omit parameters of type (), or the trivial continuation $\bot$ from an empty arity type.

For instance, the signature for the state effect, parametric in the state type s, consists of constructors *Get* and *Put* that denote operations for reading and writing the state.

```
data State s k = Get (s → k) | Put s (() → k)
```

Syntax The algebraic effect approach distinguishes between the syntax and the semantics of a monad defined by a number of algebraic operations. The syntax tree is is given by the *free monad*: a recursive structure that sequentially composes zero or more operations of the given signature.

```
data Free sig x = Var a | Op (sig (Free sig x))
```

The leaves of this tree are given by *Var* of type a, and nodes are operations *Op* that are shaped by *sig*. The result of *return x* is a leaf, and $(\gg\!=)$ grows the tree of operations at its leaves.

```
instance Functor f ⇒ Monad (Free f) where
  return x  =  Var x
  Var x    >>= f = f x
  Op op    >>= f = Op (fmap (>>= f) op)
```

The last line satisfies algebraicity (2) by construction.

Putting syntax together using these constructs directly can be cumbersome due to the additional layer of constructors. This can be alleviated by creating smart constructors:

```
get' :: Free (State s) s          put' :: s → Free (State s) ()
get' = Op (Get return)            put' s = Op (Put s return)
```

Reproducing the program *incr* at type *Free* (*State Int*) () now requires code that essentially differs only in the type:

```
incr' :: Free (State Int) ()
incr' = get' >>= put' ∘ succ
```

Semantics The syntax in a *Free sig* datatype is given a structured interpretation by a *fold* over the structure. This is thought of as a *handler* for the syntax, since the syntax in *sig* is removed, or handled away, as a result of this operation.

```
fold :: Functor sig
     ⇒ (a → b) → (sig b → b) → (Free sig a → b)
fold gen alg (Var x) = gen x
fold gen alg (Op op) = alg (fmap (fold gen alg) op)
```

The two key parameters to *fold* are the *gen*erator that interprets a values into the carrier b, and the *sig*-*alg*ebra that explains how to interpret the signature's operations. We call the triple $\langle b, gen, alg\rangle$ a *handler* for the signature *sig*.

A handler for *Free* (*State s*) a terms is $\langle s \to a, gen_S, alg_S\rangle$:

```
gen_S :: a → (s → a)
gen_S x = λs → x

alg_S :: (State s) (s → a) → (s → a)
alg_S (Get k)   = λs → k s s
alg_S (Put s k) = λ_ → k () s
```

This handler behaves in the expected way, where the following program increments and returns the state.

```
> fold gen_S alg_S (incr' >> get') 5
6
```

3.2 Modular Algebraic Effects

The modular composition of effects does not follow directly from the algebraic effects approach. Additional structure is required to allow signature and handlers to be composed.

3.2.1 Modular Signatures

Signatures compose naturally with the coproduct functor $sig_1 + sig_2$, whose neutral element is *Void*:

```
data (sig1 + sig2) a = Inl (sig1 a) | Inr (sig2 a)
data Void k
```

Any signature *sig* is isomorphic to *sig* + *Void*. This means that *Void* serves as a base case with the handler $\langle a, id, \bot\rangle$:

```
runVoid :: Free Void a → a
runVoid = fold id ⊥
```

3.2.2 Traditional Modular Handlers

There are different ways to compose handlers. Firstly, if both handlers agree on the same carrier type b and the same generator *gen*, then the coproduct mediator can be used:

```
(▽) :: (sig1 b → b) → (sig2 b → b) → ((sig1 + sig2) b → b)
(alg1 ▽ alg2) (Inl op) = alg1 op
(alg1 ▽ alg2) (Inr op) = alg2 op
```

This mediator applies the appropriate algebra depending on the operation that is present. However, this coincidence of carrier type and generator is unusual, and more machinery is required to build a more general composition scheme.

Composition of Handlers The idea is to run one handler after the other, first interpreting only the syntax in Sig_1 with an algebra alg_1 while forwarding the syntax of Sig_2 with an algebra fwd_1 and then interpreting the latter. The basic scheme for composing handlers looks like:

```
   Free (Sig1 + (Sig2 + Void)) A1   { fold gen1 (alg1 ▽ fwd1) }
→  Free (Sig2 + Void) A2            { fold gen2 (alg2 ▽ fwd2) }
→  Free Void A3
```

This scheme works if the carrier of every handler is a computation type for the remaining signature. For instance, the carrier of the first handler should be *Free* (Sig_2 + *Void*) A_2. This requirement is always implicitly met in existing languages with native support for algebraic effects and handlers like Eff [2], but in Haskell and other encodings explicit attention is required.

Consider a computation of type *Free* (*Fail* + (*State Int* + *Void*)) a where *Fail* is the signature of failing computations:

```
data Fail k = Fail

fail' = Op Fail
```

We can handle *Fail* first with a handler and generator whose carrier is *Free* (*State Int* + *Void*) (*Maybe a*), or $C_F\ a$ for short.

```
type C_F a = Free (State Int + Void) (Maybe a)

gen_MF :: a → C_F a              alg_MF :: Fail (C_F a) → (C_F a)
gen_MF x = return (Just x)       alg_MF Fail = return Nothing
```

While handling *Fail* the *State s* + *Void* operations are untouched and *forwarded* to the resulting computation. Here, the forwarding algebra that achieves this is simply *Op*:

```
fwd_MF :: (State Int + Void) (C_F a) → (C_F a)
fwd_MF op = Op op
```

This is all combined into the $handle_{MF}$ function:

$$handle_{MF} :: Free\ (Fail + (State\ Int + Void))\ a \\ \quad \to Free\ (State\ Int + Void)\ (Maybe\ a) \\ handle_{MF} = fold\ gen_{MF}\ (alg_{MF} \triangledown fwd)$$

Polymorphism for Orthogonal Effects Nothing in the definition of the *Fail* handler depends on the fact that the remaining signature is $State\ s + Void$; the handler is entirely orthogonal with respect to the remaining effect. We can expose this by making the generator and algebra signatures polymorphic in the computation monad.

$$gen_{MF} :: Monad\ m \Rightarrow a \to m\ (Maybe\ a) \\ gen_{MF}\ x = return\ (Just\ x)$$

$$alg_{MF} :: Monad\ m \Rightarrow Fail\ (m\ (Maybe\ a)) \to m\ (Maybe\ a) \\ alg_{MF}\ Fail\ \ = return\ Nothing$$

Similary, the forwarding algebra depends neither on the specific signature nor on the particular value type.

$$fwd :: Functor\ sig \Rightarrow sig\ (Free\ sig\ b) \to Free\ sig\ b \\ fwd\ op = Op\ op$$

This yields a more reusable handling function:

$$handle_{MF} :: Functor\ sig \Rightarrow Free\ (Fail + sig)\ a \\ \quad \to Free\ sig\ (Maybe\ a) \\ handle_{MF} = fold\ gen_{MF}\ (alg_{MF} \triangledown fwd)$$

Post-Processing The basic scheme does not capture all uses of modular handlers. Consider the following encoding of $State\ s$ where the carrier is $m\ (s \to m\ a)$ rather than $s \to m\ a$, i.e., with an m as the outer part of the carrier type like implicitly enforced in languages like Eff:

$$gen_{S_1} :: Monad\ m \Rightarrow a \to m\ (s \to m\ a) \\ gen_{S_1}\ x = return\ (\lambda s \to return\ x)$$

$$alg_{S_1} :: Monad\ m \Rightarrow State\ s\ (m\ (s \to m\ a)) \to m\ (s \to m\ a) \\ alg_{S_1}\ (Get\ k)\ \ \ = return\ (\lambda s \to k\ s \gg\!= \lambda f \to f\ s) \\ alg_{S_1}\ (Put\ s'\ k) = return\ (\lambda s \to k\ () \gg\!= \lambda f \to f\ s')$$

Handling the remaining effects with the basic scheme does not yield the desired result because it only affects the outer m in the carrier, and not the inner m. For this reason, effect and handler languages generally follow a more sophisticated scheme where some post-processing takes place. For instance, Eff provides a "finally" clause in its handlers in addition to the generator and the algebra. For state, this finally clause can be encoded as follows:

$$finally_S :: Monad\ m \Rightarrow s \to (m\ (s \to m\ a) \to m\ a) \\ finally_S\ s0 = \lambda p \to p \gg\!= \lambda f \to f\ s0$$

This clause runs the outer computation p and applies the resulting function f to the initial state $s0$ to run the inner computation next. This effectively collapses the two levels of computation into one level that can be handled as before.

This more general scheme can be summarized as:

$$\begin{array}{ll} \quad Free\ (Sig_1 + Sig_2 + Void)\ A_1 & \{\ fold\ gen_1\ (alg_1 \triangledown fwd_1)\ \} \\ \to C_1\ (Free\ (Sig_2 + Void)) & \{\ finally_1\ \} \\ \to Free\ (Sig_2 + Void)\ A_2 & \{\ fold\ gen_2\ (alg_2 \triangledown fwd_2)\ \} \\ \to C_2\ (Free\ Void) & \{\ finally_2\ \} \\ \to Free\ Void\ A_3 & \end{array}$$

3.2.3 Generalized Modular Handlers

In our setting, where there is freedom to break the mould of effects and handlers, the modular handling scheme can be generalized further in two ways.

Escape from the Monad Results can escape from the world of computations to yield a final carrier type:

$$\begin{array}{ll} \to Free\ Void\ A_3 & \{\ runVoid\ \} \\ \to A_3 & \end{array}$$

Generalized Carriers The carrier types do not need to be of the form $m\ A$ whith a monadic computation type m on the outside. We can for instance allow the more apt carrier type $s \to m\ a$ for $State\ s$ that does not feature the unnecessary outer m. Of course, as it is defined, the forwarding algebra is not compatible with this carrier type. Instead, we need to define a custom forwarding algebra:

$$fwd_S :: Functor\ sig \Rightarrow sig\ (s \to Free\ sig\ a) \to (s \to Free\ sig\ a) \\ fwd_S\ op = \lambda s \to Op\ (fmap\ (\lambda k \to k\ s)\ op)$$

More generally, we require that the carrier of a modular handler is of the form $c\ \ m$ where $c :: (* \to *) \to *$ is a type constructor parameterized in the type m of the remaining computation, and that it provides a forwarding algebra. These is encapsulated in the *ModularCarrier* type class:

class *ModularCarrier* c **where**
  $fwd_{MC} :: Monad\ m \Rightarrow m\ (c\ m) \to c\ m$

The forwarding algebra is formulated as a polymorphic algebra fwd_{MC} for any monad m. We can specialize it to the free monad and the form we need as follows:

$$fwd_{sig} :: (ModularCarrier\ c, Functor\ sig) \\ \quad \Rightarrow sig\ (c\ (Free\ sig)) \to c\ (Free\ sig) \\ fwd_{sig}\ op = fwd_{MC}\ (Op\ (fmap\ return\ op))$$

We require that fwd_{MC} is an Eilenberg-Moore algebra [8]. This means that it must respect the monad operations:

$$fwd_{MC} \circ return = id \quad (3)$$

$$fwd_{MC} \circ join = fwd_{MC} \circ fmap\ fwd_{MC} \quad (4)$$

The first law states that nothing happens when forwarding *return*, while the second states that forwarding a sequence of steps as a batch is the same as forwarding them individually.

Summary In summary, we define a modular handler for a signature S from values of type A to type B as the quadruple[1] $\langle C, gen, alg, finally\rangle$ where $C :: (* \to *) \to *$ is a modular carrier, and all three of the generator $gen :: Monad\ m \Rightarrow A \to C\ m$, algebra $alg :: Monad\ m \Rightarrow S\ (C\ m) \to C\ m$, and finally function $finally :: Monad\ m \Rightarrow C\ m \to m\ B$ are polymorphic in the monad m.

The general handling scheme is then of the form:

$$\begin{array}{ll} & Free\ (Sig_1 + Sig_2 + Void)\ A_1 & \{\ fold\ gen_1\ (alg_1 \triangledown fwd_{sig})\ \} \\ \to & C_1\ (Free\ sig_2 + Void) & \{\ finally_1\ \} \\ \to & Free\ (Sig_2 + Void)\ A_2 & \{\ fold\ gen_2\ (alg_2 \triangledown fwd_{sig})\ \} \\ \to & C_2\ (Free\ Void) & \{\ finally_2\ \} \\ \to & Free\ Void\ A_3 & \{\ runVoid\ \} \\ \to & A_3 & \end{array}$$

Example To illustrate the approach, we wrap the $State\ s$ carrier above it in a newtype constructor:

```
newtype StateH s a m = StateH { runStateH :: s → m a }
```

This way we can formulate a *ModularCarrier* instance:

```
instance ModularCarrier (StateH s a) where
  fwdMC mf = StateH (λs → do f ← mf; runStateH f s)
```

Now the semantics of $State$ can be given as follows:

```
genSH :: Monad m ⇒ a → StateH s a m
genSH x = StateH (λs → return x)

algSH :: Monad m ⇒ State s (StateH s a m) → StateH s a m
algSH (Get k)   = StateH (λs → runStateH (k s) s)
algSH (Put s k) = StateH (λ_ → runStateH (k ()) s)
```

With an appropriate finally function we get the modular $State\ Int$ handler $\langle StateH\ Int\ Int, gen_{SH}, alg_{SH}, flip\ runStateH\ 5\rangle$:

```
> (runId ∘ flip runStateH 5 ∘ fold genSH algSH) (incr' ≫ get')
6
```

We can compose this modular $StateH\ s\ a$ with other effects such as failure introduced above. A modular handler is $\langle MaybeH\ a\ m, gen_F, alg_F, runMaybe_H\rangle$, where the modular carrier is wrapped in a newtype for the type class instance:

```
newtype MaybeH a m =
  MaybeH { runMaybeH :: m (Maybe a) }

instance ModularCarrier (MaybeH a) where
  fwdMC mf = MaybeH (do f ← mf; runMaybeH f)
```

Observe that $MaybeH$ uses the same representation as $MaybeT$, but has its type parameters swapped; in Section 6.2 we show that other monad transformers give rise to modular carriers in a similar manner.

The associated generator and algebra are:

```
genF :: Monad m ⇒ a → MaybeH a m
genF x = MaybeH (return (Just x))

algF :: Monad m ⇒ Fail (MaybeH a m) → MaybeH a m
algF Fail = MaybeH (return Nothing)
```

Now putting the pieces together, we can work with composition in whichever way we want.

```
hdlFS :: s → Free (Fail + (State s + Void)) a → Maybe a
hdlFS s = runVoid ∘
          flip runStateH s ∘ fold genSH (algSH ▽ fwdsig) ∘
          runMaybeH ∘ fold genF (algF ▽ fwdsig)
hdlSF :: s → Free (State s + (Fail + Void)) a → Maybe a
hdlSF s = runVoid ∘
          runMaybeH ∘ fold genF (algF ▽ fwdsig) ∘
          flip runStateH s ∘ fold genSH (algSH ▽ fwdsig)
```

While the above two handlers for state and failure are both polymorphic in the value type a, this is not a necessity for handlers. Consider the following alternative handler $\langle Def\ m, gen_D, alg_D, runDef\rangle$ for failure that only works for Int computations.

```
newtype Def m = Def { runDef :: m Int }

instance ModularCarrier Def where
  fwdMC mf = Def (do f ← mf; runDef f)

genD :: Monad m ⇒ Int → Def m
genD x = Def (return x)

algD :: Monad m ⇒ Fail (Def m) → Def m
algD Fail = Def (return 0)
```

This handler does not abort the computation upon failure, but instead proceeds with the default value 0. Section 5 shows how to formulate similar monad transformers that can only be run with computations of a particular value type.

4 Comparing the Two Approaches

The remainder of this paper compares the expressivity of the two approaches. We start by highlighting some of the differences between the two.

Overloadable Syntax Both approaches provide a monadic syntax for effectful computations that can be overloaded with different semantics. In the case of monad transformers this overloadable syntax is captured in terms of a polymorphic type m that is constrained by monad subclass constraints. For the state effect we use the type

$$MonadState\ s\ m \Rightarrow m$$

Note that the constraint polymorphism not only leaves the semantics open, but also whether other effects can be used in the computation. In order to specify that multiple effects are combined, we pile up monad subclass constraints. For instance, the constraint

$$(MonadState\ s\ m, MonadFail\ m) \Rightarrow m$$

expresses that both the state and failure effects can be used.

[1] We often do not explicitly identify the *finally* function, in particular when it is a trivial newtype isomorphism.

In the case of algebraic effects, we use the free monad for the syntax of monadic computations. It is parameterised in the signature functor of the particular effects that can occur. Multiple effects are combined using the (functor) coproduct of the signatures. For instance, the type

$$Free\ (\textsc{State}\ s + \textsc{Fail})$$

provides state and failure effects. The type that specifies that the state effect can occur with other effects is

$$Functor\ sig \Rightarrow Free\ (\textsc{State}\ s + sig).$$

Because the monad transformer approach uses type-class constraints to overload syntax, the order in which they are written does not matter. E.g., (*MonadState s m*, *MonadFail m*) is the same constraint as (*MonadFail m*, *MonadState s m*). On the other hand, the algebraic-effects approach seems less flexible, since the coproduct fixes an order on signatures. However, this is not an essential shortcoming, as there are implementation techniques that abstract away the exact order of the coproduct [23, 40].

Expressivity of effect manipulating functions In the monad transformers approach, effects are manipulated by member functions of a monad subclass. In principle, there are no limitations on what a member function can be, but this freedom is a double-edged sword: the functions have no structure and therefore it is difficult to solve the problem of lifting a function of a monad to the transformed monad. An important contribution of Liang et al. [27] was to show how this is possible for a range of important effects.

In the algebraic effects approach, effects are introduced with algebraic operations as determined by a signature. Algebraic operations for a signature functor $\textsc{Sig}$ correspond to functions $\forall a.\ \textsc{Sig}\ a \to M\ a$. This restriction on the type of the operations provides more structure, but leaves out operations which work over a scope, such as the exception-handling operation *catch*. The problem of expressing scoping operations led Plotkin and Pretnar [34] to introduce the notion of handlers. However, Wu et al. [44] showed that treating scoping operations as handlers causes modularity problems, since these operations tie together syntax and semantics, and therefore some semantics cannot be expressed without changing the original program (see discussion below).

Algebraic operations can be easily lifted through a monad transformer by post-composing with *lift*. An advantage of the modular algebraic approach is that handlers only need to deal with the topmost effect, whereas monad transformers would require a special lifting function. Therefore, there seems to be a trade-off: either liftings for scoped operations such as *catch* are provided, or some modularity is lost.

Effect Semantics Both approaches provide a way to assign different semantics to the same syntax.

In the case of monad transformers, the semantics are assigned by instantiating the polymorphic type variable *m* with a stack of transformers that satisfies all the type class constraints. The variation in semantics is possible because there are different stacks that satisfy the same set of constraints.

Firstly, transformers can be reordered, which gives rise to different interactions between the effects. For instance, both $State_T\ s\ (Maybe_T\ Id)$ and $Maybe_T\ (State_T\ s\ Id)$ satisfy the constraints (*MonadState s m*, *MonadFail m*), but give rise to different interactions between state and failure handling.

Secondly, multiple transformers can satisfy the same constraint. For instance, the logging state transformer also satisfies *MonadState s m* and records the intermediate states.

$$\begin{aligned}
&\textbf{newtype}\ LogState_T\ s\ m\ a = \\
&\quad LST\ \{runLST :: s \to m\ (a, s, [s])\} \\
&\textbf{instance}\ Trans\ (LogState_T\ s)\ \textbf{where} \\
&\quad lift\ m = LST\ (\lambda s \to \textbf{do}\ x \leftarrow m; return\ (x, s, [\,]))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ Monad\ m \Rightarrow Monad\ (LogState_T\ s\ m)\ \textbf{where} \\
&\quad return\ x = LST\ (\lambda s \to return\ (x, s, [\,])) \\
&\quad m \gg\!= f\ \ = LST\ (\lambda s \to \textbf{do}\ (x, s', h_1)\ \leftarrow runLST\ m\ s \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (y, s'', h_2) \leftarrow runLST\ (f\ x)\ s' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (y, s'', h_1 +\!\!+ h_2))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ Monad\ m \Rightarrow MonadState\ s\ (LogState_T\ s\ m)\ \textbf{where} \\
&\quad get\ \ \ = LST\ (\lambda s\ \to return\ (s, s, [\,])) \\
&\quad put\ s = LST\ (\lambda s' \to return\ ((), s, [s']))
\end{aligned}$$

The instance that is used is decided by specifically stating the desired concrete type.

In the algebraic-effect approach, the handlers are in charge of assigning semantics by folding the syntax tree into a particular carrier by means of a particular algebra. By using different handlers to interpret the same effect, we obtain flexibility in the interpretation. For instance, the following handler logs state operations:

$$\begin{aligned}
&\textbf{newtype}\ LogState_H\ s\ a\ m \\
&\quad = LS\ \{runLS_H :: s \to [s] \to m\ (a, [s])\} \\
&gen_{LS} :: Monad\ m \Rightarrow a \to LogState_H\ s\ a\ m \\
&gen_{LS}\ x = LS\ (\lambda s\ h \to return\ (x, reverse\ h)) \\
&alg_{LS} :: \textsc{State}\ s\ (LogState_H\ s\ a\ m) \to LogState_H\ s\ a\ m \\
&alg_{LS}\ (\textsc{Get}\ k)\ \ \ = LS\ (\lambda s\ \ h \to runLS_H\ (k\ s)\ \ s\ h) \\
&alg_{LS}\ (\textsc{Put}\ s\ k) = LS\ (\lambda s'\ h \to runLS_H\ (k\ ())\ s\ (s' : h))
\end{aligned}$$

Similar to the transformer approach, we may also control the effect interaction by running the handlers in different orders. (In our crude implementation, we would also need to re-arrange the order of the coproduct of signatures to match the order of handlers.) However, there is a catch: since effect manipulating functions such as *catch* are handlers, a program may have handlers interspersed with algebraic operations. Consequently, the ordering of effects may be determined by the structure of the program, and some interpretations may be impossible to achieve for such a program [44].

Despite these differences, monad transformers and effect handlers have much in common. In the remainder of this

paper we formalise these similarities by showing a class of transformers that correspond to effect handlers, and vice versa. This establishes that there is a significant common ground between the two approaches.

5 Algebraic Effects as Monad Transformers

This section establishes that the algebraic effects approach is definitely not more expressive than the transformers approach by embedding the former in the latter. In particular, we show how to generically derive a monad transformer from an algebraic effect.

5.1 From Signature to Monad Subclass

Whereas in the algebraic effects approach operations are characterised by a signature functor, monad transformers provide operations via a monad subclass. A monad subclass is defined in terms of a given *sig*nature functor:

```
class (Monad m, Functor sig) ⇒ MonadEff sig m | m → sig
  where eff :: sig x → m x
```

This class declaration is generic in the signature and provides exactly one algebraic operation *eff* that distinguishes between an effect's different operations by taking the syntax of the desired operation as a parameter.

Here we have not used the type $sig\ (m\ a) \to m\ x$ for operations introduced in Section 3.1. Instead, we have used an alternative representation that enforces algebraicity (2) by the type system alone rather than with an additional law.

The correspondence between the two presentations of algebraic operations is as follows:

```
algEff :: MonadEff sig m ⇒ sig (m x) → m x
algEff = join ∘ eff

fromAlg :: (Functor sig, Monad m)
        ⇒ (∀x. sig (m x) → m x) → (sig x → m x)
fromAlg op = op ∘ fmap return
```

This isomorphism can be shown by proving one way that $fromAlg\ algEff = eff$, and the other way, assuming that $eff = fromAlg\ op$ and that *op* is algebraic, then $algEff = op$.

There is a trivial instance of *MonadEff* which interprets any signature in the corresponding free monad.

```
instance Functor sig ⇒ MonadEff sig (Free sig) where
  eff op = Op (fmap return op)
```

This places a *return* at every continuation.

Alternative Signatures Neither the form $sig\ x \to m\ x$ nor $sig\ (m\ x) \to m\ x$ is always the most convenient for operations. We can often provide more convenient operations, by using so-called generic effects [33].

Using the functor $Sig\ a\ b$ (Section 3.1) gives the following equivalence [19]. For all functors *m*, and types *a* and *b*,

$$a \to m\ b \quad \cong \quad \forall x.\ Sig\ a\ b\ x \to m\ x \tag{5}$$

The components of the isomorphism are as follows:

```
toSig :: Functor m ⇒ (a → m b) → (∀x. Sig a b x → m x)
toSig f (Op a g) = fmap g (f a)

fromSig :: Functor m ⇒
  (∀x. Sig a b x → m x) → (a → m b)
fromSig op a = op (Op a id)
```

When a signature functor is isomorphic to $Sig\ A\ B$ for some *A* and *B*, then we can use the equivalence above and obtain a generic effect. (This is an application of the Yoneda lemma.)

For example, for the state effect, the *MonadState* interface of Section 2.2 can be recovered; $State\ s$ is isomorphic to $Sig\ ()\ s + Sig\ s\ ()$, and so equivalence (5) gives two operations:

```
get :: MonadEff (State s) m ⇒ () → m s
get () = eff (Get id)

put :: MonadEff (State s) m ⇒ s → m ()
put s = eff (Put s id)
```

Note that, if we specialize these definitions to the free monad, we get the smart constructors in Section 3.1.

Lifting *eff* of the base monad through a monad transformer *T* is simply post-composition with *lift*:

```
instance MonadEff sig m ⇒ MonadEff sig (T m) where
  eff op = lift ∘ eff
```

Therefore, the lifting of algebraic operations through any monad transformer is completely unproblematic and an implementation could provide it automatically.

5.2 From Handler to Monad Transformer

Now that we have a suitable monad subclass, we can provide a monad transformer that instantiates this subclass in terms of a given handler. In fact, we give two ways to do so below.

The Free Transformer The free monad transformer over a given signature *sig* is a generic monad transformer that instantiates *MonadEff*.

```
newtype FreeT sig m a = FreeT { runF :: m (FreeF sig m a) }
data FreeF sig m a = ReturnF a | OpF (sig (FreeT sig m a))
```

Its monad instance is analogous to the free monad, except that it interleaves operations with the transformed monad.

```
instance (Monad m, Functor sig)
  ⇒ Monad (FreeT sig m) where
    return x          = FreeT (return (ReturnF x))
    (FreeT t) >>= f = FreeT (t >>= go) where
      go (ReturnF a) = runF (f a)
      go (OpF op)    = return (OpF (fmap (>>=f) op))
```

The transformed monad $FreeT\ sig_1\ m$ implements both the operations from the signature sig_1 and the algebraic operations of the monad *m*. Thus, we obtain the following instance:

```
instance (Functor sig1, MonadEff sig2 m)
  ⇒ MonadEff (sig1 + sig2) (Free_T sig1 m) where
    eff (Inl op) = Free_T (return (Op_F (fmap return op)))
    eff (Inr op) = lift (eff op)
```

This definition clearly does not rely on the handler at all. All the work to actually interpret the syntax and recover the handler's semantics is in the corresponding fold function.

```
foldFree_T :: (Monad m, Functor sig, ModularCarrier c)
           ⇒ (a → c m) → (sig (c m) → c m)
           → Free_T sig m a → c m
foldFree_T gen alg = goM where
  goM = fwd_MC ∘ fmap goSig ∘ run_F
  goSig (Return_F x) = gen x
  goSig (Op_F op)    = alg (fmap goM op)
```

For all signatures sig_1 and sig_2, it is the case that $Free\ (sig_1 + sig_2)$ is isomorphic to $Free_T\ sig_1\ (Free\ sig_2)$. The isomorphism is given as follows:

```
toFree_T :: (Functor sig1, Functor sig2)
    ⇒ Free (sig1 + sig2) a → Free_T sig1 (Free sig2) a
toFree_T = fold return (join ∘ eff)

fromFree_T :: (Functor sig1, Functor sig2)
    ⇒ Free_T sig1 (Free sig2) a → Free (sig1 + sig2) a
fromFree_T =
  join ∘ fold return (join ∘ eff ∘ Inr) ∘ fmap fromFree_F ∘ run_F
    where fromFree_F (Return_F a) = return a
          fromFree_F (Op_F op)    =
            join (eff (Inl (fmap fromFree_T op)))
```

The functions above are mutual inverses, but they are also monad morphisms. (A category-theory inclined reader will recognise this as a consequence of Hyland et al.'s [15] characterisation of $Free_T\ sig\ m$ as a coproduct of m and $Free\ sig$ in the category of monads and monad morphisms.) Moreover, the respective folds correspond:

$$fold\ gen\ (liftAlg\ alg) = foldFree_T\ gen\ alg \circ toFree_T$$

Finally, $Free_T$'s run function puts everything together:

```
runFree_T :: (Monad m, Functor sig, ModularCarrier c)
  ⇒ (a → c m) → (sig (c m) → c m) → (c m → m b)
  → Free_T sig m a → m b
runFree_T gen alg finally = finally ∘ foldFree_T gen alg
```

The Non-Free Transformer The above definition is not entirely satisfactory because it relies on $Free_T$ as an intermediate data structure. Instead, we may want a transformer that captures the intended denotation c in its carrier type. We obtain this with a instance of the continuation monad.

```
newtype Cont r x = Cont {runCont :: (x → r) → r}
```

We specialise *Cont* so that the return type is a modular carrier, and the monad instance for *ContC* is essentially the same as the well-established one for *Cont*.

```
newtype ContC c (m :: ∗ → ∗) a =
  ContC {unContC :: (a → c m) → c m}

instance Monad (ContC c m) where
  return x = ContC (λk → k x)
  m >>= k  =
    ContC (λc → unContC m (λx → unContC (k x) c))
```

For any modular carrier c, $ContC\ c$ is a monad transformer.

```
instance ModularCarrier c ⇒ Trans (ContC c) where
  lift m = ContC (λk → fwd_MC (fmap k m))
```

For any fixed signature *SigC*, modular carrier *C*, and algebra $algC :: SigC\ (C\ m) \to C\ m$, we may define *ContC C m* to be a signature monad, using the following generic function:

```
effContC :: (Functor sig)
         ⇒ (sig (c m) → c m) → sig x → ContC c m x
effContC alg s = ContC (λk → alg (fmap k s))

instance (MonadEff sig2 m)
  ⇒ MonadEff (SigC + sig2) (ContC C m) where
    eff (Inl op) = effContC algC op
    eff (Inr op) = lift (eff op)
```

We embed syntax in $ContC\ c$ using $toContC\ alg$. Unlike the transformation into the free monad transformer, the application of $toContC\ alg$ is effectively giving semantics to the syntax, and therefore there is no way back.

```
toContC :: (Functor sig1, Functor sig2, ModularCarrier c)
        ⇒ (∀m. Monad m ⇒ sig1 (c m) → c m)
        → Free (sig1 + sig2) a → ContC c (Free sig2) a
toContC alg =
  fold return ((join ∘ effContC alg) ▽ (join ∘ lift ∘ eff))
```

The correctness of this embedding lies in the fact that for all algebras *alg*, the function $toContC\ alg$ is a monad morphism, and that any handler $fold\ gen\ (liftAlg\ alg)$ can be recovered as the following composition:

$$fold\ gen\ (liftAlg\ alg) = unContC\ gen \circ toContC\ alg$$

Again, the run function puts everything together:

```
runContC :: (Monad m, Functor sig, ModularCarrier c)
         ⇒ (a → c m) → (c m → m b)
         → ContC c m a → m b
runContC gen finally = finally ∘ unContC gen
```

Example Let us illustrate the second approach on the *Def* handler from Section 3.2.3 and derive a transformer Def_T. Specializing the definitions makes its code more palatable.

```
newtype Def_T m a =
  Def_T {unDef_T :: (a → m Int) → m Int}
```

The specialised *MonadEff* instance is a *MonadFail* instance:

```
instance Monad m ⇒ MonadFail (Def_T m) where
  fail = Def_T (λk → return 0)
```

The run function is rather atypical for monad transformers in that it only applies to computations that yield an *Int*, rather than computations of any type.

```
runDef_T :: Monad m ⇒ Def_T m Int → m Int
runDef_T (Def_T p) = p return
```

This specificity is not unusual for handlers, and our construction shows that it carries over easily to transformers.

6 Monad Transformers as Algebraic Effects

Given a monad subclass *MonadX* and corresponding transformer T that implements the operations of *MonadX*, we need to distinguish between the algebraic operations, and the non-algebraic ones.

The lack of structure in the type signatures of member functions of the monad sub-classes greatly complicates a systematic translation from monad transformers into the more structured approach of modular algebraic effects. Nevertheless, it is often possible to systematically identify algebraic operations using the equivalences shown in Section 5.1.

6.1 Transformer Signature

As discussed in Section 5.1, algebraic operations come in many different guises. However, in the monad-transformer approach, they are usually presented as a generic effect:

$$op :: A \to m\ B$$

where A and B are types that do not contain the type variable m. Using equivalence (5), we obtain that the signature functor for such an operation must be $Sig\ A\ B$. For instance, this way we can derive the standard *State* signature (Section 3.1) from the *MonadState* class.

Any type variables that are universally quantified and occur only in the return type of a generic effect can be interpreted as a nullary operation. For instance, this happens to the type variable a of the *mzero* method in *MonadPlus*:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

The type of *mzero* is, explicitly, $\forall a.\ Monad\ m \Rightarrow m\ a$. The universal quantification implies that an a value is never produced, and therefore it can be replaced by an empty type:

```
data Void
```

which has the property that for every type r there is exactly one inhabitant of type $Void \to r$. Changing the type $\forall a.\ Monad\ m \Rightarrow m\ a$ to $Monad\ m \Rightarrow m\ Void$, and adding a unit input gives the following equivalent type for *mzero*:

```
mzero :: () → m Void
```

The type of *mzero* gives rise to its signature functor:

```
data NondetF k = MzeroF () (Void → k) | ...
```

Since both the unit type () and the type $(Void \to r)$ are trivial they can be removed.

```
data NondetF k = MzeroF | ...
```

In contrast, the signature of $mplus :: m\ a \to m\ a \to m\ a$ is not that of a generic effect. It is directly given as an operation with two continuation parameters. To be algebraic, it must of course also satisfy the algebraicity property. This is indeed the case for most of the usual instances [36], except for *Maybe*. With this restriction, the operation *mplus* is algebraic with signature *SIG* () *Bool*, or, in specialised form,

```
data ChooseF k = ChooseF (Bool → k)
```

It is equivalent to a generic effect $choose :: () \to m\ Bool$ that returns one of two possibilities.

Incorporating this into *NondetF* by renaming the constructor, results in a signature functor for *MonadPlus*:

```
data NondetF k = MzeroF | MplusF (Bool → k)
```

This is, of course, isomorphic to $Fail + Choose$, we can give it a semantics with the handler $\langle MPlus_H\ m\ a, gen_{MP}, alg_{MP}\rangle$:

```
newtype NonDetH a m = NonDetH {runNonDetH :: m [a]}
instance ModularCarrier (NonDetH a) where
  fwd_MC = NonDetH ∘ join ∘ fmap runNonDetH
genNonDetH :: Monad m ⇒ a → NonDetH a m
genNonDetH x = NonDetH (return [x])
algNonDetH :: Monad m ⇒
  NondetF (NonDetH a m) → NonDetH a m
algNonDetH (MplusF k) =
  NonDetH (do xs ← runNonDetH (k True)
              ys ← runNonDetH (k False)
              return (xs ++ ys))
algNonDetH (MzeroF) = NonDetH (return [])
```

As fwd_{MC} is just *join*, $MPlus_H\ a$ is clearly a modular carrier.

6.2 Transformer Handler

It is easy to write a modular handler for a monad transformer T: the carrier type of the handler is the transformer type itself, as long as we fix the return type:

```
newtype Wrap_T a m = Wrap_T {unWrap_T :: T m a}
```

The monadic structure readily provides implementations for the operations and for the generator. In detail, if T implements a signature Sig_T via a function $eff_T :: Monad\ m \Rightarrow Sig_T\ a \to T\ m\ a$, an algebra and generator are defined as:

```
alg_T :: (Monad m) ⇒ Sig_T (Wrap_T a m) → Wrap_T a m
alg_T = Wrap_T ∘ join ∘ eff_T ∘ fmap unWrap_T
gen_T :: Monad m ⇒ a → Wrap_T a m
gen_T = Wrap_T ∘ return
```

The transformer T also makes for a modular carrier:

instance *ModularCarrier* ($Wrap_T$ a) **where**
  $fwd_{MC} = Wrap_T \circ join \circ lift \circ fmap\ unWrap_T$

With this, we define the following handler:

$hdl_T :: Free\ (Sig_T + sig)\ a \to Wrap_T\ a\ (Free\ sig)$
$hdl_T = Wrap_T \circ fold\ gen_T\ (liftAlg\ alg_T)$

This embedding is correct because $unWrap_T\ hdl_T$ is a monad morphism that respects the eff_T function in the sense that

$$eff_T \triangledown eff = unWrap_T \circ hdl_T \circ eff$$

where *eff* in the left-hand side of the equation comes from the *MonadEff* instance of the *Free sig* type, *eff* in the right-hand side comes from the instance for *Free* (Sig_T + *sig*). The fact above follows from the universal property of free monads. It states that for all signatures (functors) S, monads M, and polymorphic functions (natural transformations) $f :: S\ a \to M\ a$, it is the case that *fold return f* is a monad morphism, and the following holds for all monad morphisms m, where *eff* comes from the *MonadEff* instance of the *Free S* type.

$$m = fold\ return\ f \quad \Longleftrightarrow \quad m \circ eff = f$$

Finally, given $run_T :: Monad \Rightarrow T\ m\ A \to m\ B$, an appropriate finally function for the handler can be derived:

$finally :: Monad\ m \Rightarrow Wrap_T\ A\ m \to m\ B$
$finally = run_T \circ unWrap_T$

Thus it is possible to systematically obtain algebraic operations and a handler from a monad transformer restricted to its algebraic operations. Unfortunately, for the non-algebraic operations of a transformer it needs to be evaluated on a case-by-case basis whether they have algebraic counterparts. This is explored with an example of *callCC* in the next section.

7 Case Study: Call/CC

This section investigates how to express the well-known call-with-current-continuation operation *callCC* with both monad transformers and effect handlers.

7.1 Established Implementation

The MTL monad transformers library features an established interface of *callCC* in the form of the *MonadCont* type class:

class *Monad* $m \Rightarrow$ *MonadCont* m **where**
  $callCC :: ((a \to m\ b) \to m\ a) \to m\ a$

which is implemented by the continuation monad *Cont* r:

instance *MonadCont* (*Cont* r) **where**
  $callCC\ f = Cont\ (\lambda k \to$
    $runCont\ (f\ (\lambda x \to Cont\ (const\ (k\ x))))\ k)$

This implementation has been generalised to a monad transformer in a straightforward way:

newtype $ContT\ r\ m\ a = CT\ \{runCT :: (a \to m\ r) \to m\ r\}$

instance *Monad* $m \Rightarrow$ *MonadCont* ($ContT\ r\ m$) **where**
  $callCC\ f = CT\ (\lambda k \to$
    $runCT\ (f\ (\lambda x \to CT\ (const\ (k\ x))))\ k)$

At first sight, it would seem that an operation could not be further from algebraic than *callCC*. The main difficulty is that the parameter a appears in both positive and negative position: it occurs both in the domain and codomain of functions. This makes *callCC* rather unsuitable for the effect handlers approach: at first glance it seems that monad transformers are more expressive on this account.

7.2 Reformulation

Nevertheless, following Thielecke [41] and Fiore and Staton [10], we can decompose *callCC* into two algebraic operations, given by the following *MonadJump* type class:

class *Monad* $m \Rightarrow$ *MonadJump* $ref\ m \mid m \to ref$ **where**
  $jump :: ref\ a \to a \to m\ b$
  $sub\ \ :: (ref\ a \to m\ b) \to (a \to m\ b) \to m\ b$

Here $ref\ a$ is the type of a reference to a computation that takes a value of type a as input. The *jump* operation abandons the current continuation and instead runs the referenced computation with the given input. The computation *sub p q* constructs a reference out of the alternative computation q and then runs the main computation p with this reference. This characterisation is captured in the following four laws,

$$sub\ (\lambda r \to jump\ r\ x)\ k \equiv k\ x$$
$$sub\ (\lambda\_ \to p)\ k \equiv p$$
$$sub\ p\ (jump\ r') \equiv p\ r'$$
$$sub\ (\lambda r_1 \to sub\ (\lambda r_2 \to p\ r_1\ r_2)\ (k_2\ r_1))\ k_1 \equiv$$
$$sub\ (\lambda r_2 \to sub\ (\lambda r_1 \to p\ r_1\ r_2)\ k_1)\ (sub\ k_2\ k_1)$$

in addition to the already informally stated requirement that *jump* and *sub* are algebraic:

$$jump\ r\ x \gg\!= k \equiv jump\ r\ x$$
$$sub\ p\ q \gg\!= k \equiv sub\ (p \gg\!> k)\ (q \gg\!> k)$$

The former expresses that a *jump* abandons the current continuation k. The latter expresses that both the main computation and the alternative computation share the common continuation k.

Encoding *callCC* We can express *callCC* in terms of *jump* and *sub* as follows.

$callCC\ f = sub\ (\lambda ref \to f\ (jump\ ref))\ return$

Here the *exit* mechanism is made explicit by *jump*, which jumps to $return \gg\!> k \equiv k$, with the current continuation k.

Encoding *jump* ***and*** *sub* Vice versa, we can also express *jump* and *sub* in terms of *callCC*.

newtype $Ref\ m\ a = \forall r.\ R\ \{unRef :: a \to m\ r\}$

$jump\ (R\ exit)\ x = exit\ x \gg\!> return\ \bot$

$$sub\ k_1\ k_2 \quad = callCC\ (\lambda exit \to k_1\ (R\ (k_2 \gg\!\!> exit)))$$

Here we represent a reference to an alternative computation by an actual computation that performs this jump, wrapped in the newtype *Ref*. Hence, *jump*ing consists of unwrapping the newtype and running the computation, followed by an unreachable *return* $\bot$ to obtain an arbitrary return type. The *sub* operation grabs the current continuation *exit* by means of *callCC*, prefixes it with the alternative k_2, wraps it in the newtype and hands it off to the main computation k_1.

The two encodings are mutual inverses. One direction of this property is established with straightforward equational reasoning, assuming a standard *callCC* property [5],

$$callCC\ f = callCC\ (\lambda exit \to f\ (\lambda x \to exit\ x \gg\!= k))$$

which states that *exit* never returns.

The other direction of the proof is more involved and requires the techniques developed by Thielecke [41] and by Fiore and Staton [10].

7.3 Alternative Handler for *callCC*

The algebraic operations *jump* and *sub* have the signature:

$$\begin{aligned}\textbf{data}\ \textsc{Subst}\ ref\ k = {} & \forall a.\ \textsc{Jmp}\ (ref\ a)\ a \\ & \mid \forall a.\ \textsc{Sub}\ (ref\ a \to k)\ (a \to k)\end{aligned}$$

for which we can easily derive a modular handler from the *ContC* monad transformer definition following the recipe of Section 6. However, we can also provide a more direct alternative implementation that does not require an existing implementation of *callCC*. The carrier of this direct handler is the trivial identity carrier Id_H.

$$\textbf{newtype}\ Id_H\ a\ m = Id_H\ \{runId_H :: m\ a\}$$

The key idea of the handler is to represent references of type *ref a* as functions $a \to Id_H\ r\ m$. We want the type *ref a* to be functorial in the type *a*, but in Haskell functoriality is always in the last argument of a parameterized type. To work around this limitation, we represent $a \to Id_H\ r\ m$ by the newtype alias $Id_H\ r\ m :\!\leftarrow a$, where *a* has been exposed as the last parameter.

$$\textbf{newtype}\ b :\!\leftarrow a = Switch\ \{(\$) :: a \to b\}$$

The constructor $Switch :: (a \to b) \to (b :\!\leftarrow a)$ switches the direction of a function, and the deconstructor $(\$) :: (b :\!\leftarrow a) \to a \to b$ is reminiscent of Haskell's function application $(\$) :: (a \to b) \to a \to b$.

With this choice of representation, the handler's algebra and generator are trivial.

$$\begin{aligned}&gen_{SB} :: Monad\ m \Rightarrow r \to Id_H\ r\ m \\ &gen_{SB}\ x = Id_H\ (return\ x) \\ &alg_{SB} :: Monad\ m \Rightarrow \\ &\quad \textsc{Subst}\ ((:\!\leftarrow)\ (Id_H\ r\ m))\ (Id_H\ r\ m) \to Id_H\ r\ m \\ &alg_{SB}\ (\textsc{Jmp}\ ref\ x) = ref \$ x \\ &alg_{SB}\ (\textsc{Sub}\ k_1\ k_2) = k_1\ (Switch\ k_2)\end{aligned}$$

Note that in the type of the algebra, neither the value type *r* nor the monad type *m* is orthogonal with respect to the functor's type $\textsc{Subst}\ ((:\!\leftarrow)\ (Id_H\ r\ m))$. Hence, when we use the free monad transformer recipe of Section 6, this non-orthogonality carries over to the *run* function:

$$\begin{aligned}&runSubstT :: Monad\ m \\ &\quad \Rightarrow Free_T\ (\textsc{Subst}\ ((:\!\leftarrow)\ (Id_H\ r\ m)))\ m\ r \to Id_H\ r\ m \\ &runSubstT = foldFree_T\ gen_{SB}\ alg_{SB}\end{aligned}$$

We can however institute orthogonality with respect to the value type by means of an additional continuation argument.

$$\begin{aligned}&runSubstT' :: Monad\ m \\ &\quad \Rightarrow Free_T\ (\textsc{Subst}\ ((:\!\leftarrow)\ (Id_H\ r\ m)))\ m\ a \\ &\quad \to ((a \to m\ r) \to Id_H\ r\ m) \\ &runSubstT'\ p\ k = runSubstT\ (p \gg\!= lift \circ k)\end{aligned}$$

This brings us essentially back to the continuation monad transformer as $((a \to m\ r) \to Id_H\ m\ r) \cong ContT\ r\ m\ a$.

8 Related Work

8.1 Algebraic Effects and Handlers

Plotkin and Power [32] were the first to explore effect operations, and gave an algebraic account of effects [33] and their combination [15]. Subsequently, Plotkin and Pretnar [35] have added the concept of handlers to deal with exceptions. This theoretical development has led to many language and library implementations. Unlike our work, these implementations generally restrict themselves to a call-by-push-value setting [26] where handler clauses yield computations.

Eff Perhaps the most prominent language is Eff [2], an OCaml-like language with native support for algebraic effects and handlers. An implementation that embeds Eff directly into OCaml with delimited continuations has also been developed [24]. Eff does not feature an explicit free monad datatype, but distinguishes between syntactic sorts for (possibly effectful) computations and pure values. Eff only supports handler carrier types of the form (using our terminology) *Free F A*; carriers of the form $S \to Free\ F\ B$ are encoded as $Free\ F\ (S \to Free\ F\ B)$. A special case are handlers that introduce new effects in the program, i.e., of type $Free\ (F + G)\ A \to Free\ (H + G)\ B$. We can model their modular carrier as $Free_T\ H$ since $Free_T\ H\ (Free\ G) \cong Free\ (H + G)$.

While Eff implicitly forwards operations that are not explicitly handled, its subtyping-based type system [1] does not allow the characterisation of modular carriers. Nevertheless many of its example handlers fall in this class. Eff also allows a class of semi-modular handlers that rely on the presence of another effect, which can be expressed as relaxing $(Monad\ m) \Rightarrow m$ to $(MonadEff\ F\ m) \Rightarrow m$.

Handlers in Action Handlers in Action [21] builds on a formalisation similar to Eff's with carrier types essentially of the form *Free F A* and a simple type system. It

comes with an implementation in Haskell, among other functional languages, whose Template Haskell front-end syntax supports *open* (i.e., essentially modular) handlers and exploits Haskell's polymorphism to encode them. In addition to *Free F A*, this implementation also supports carrier types of the form $S \to Free\ F\ A$. They also introduce *shallow* handlers that only handle the first operation. We can model these as folds with tupling [14].

The implementation employs a *final* encoding [4] of the free monad syntax where a conjunction of type class constraints implements the coproduct construction and provides various ways to express handlers, either as explicit *fold*s and *build*s or in fused form. An explanation of fusion for free monads can be found in the work of Wu and Schrijvers [43].

Extensible Effects The Extensible Effects library [23] has arrived at essentially the same functionality, but does not attribute the algebraic effects and handlers theory as its original inspiration. This work puts much emphasis on efficient representation of the free monad and the functor co-product. More recently Kiselyov and Ishii [22] used an inlined co-Yoneda construction and a queue datatype [31] in order to improve efficiency. The library provides modular *fold* recursion schemes for $Free\ F\ A$ and $S \to Free\ F\ A$. Moreover, it replicates much of the functionality of the MTL monad transformers library in terms of modular handlers.

Other Implementations Idris provides an effect handlers library [3] based on the indexed free monad with built-in co-Yoneda construction which represents the signature coproduct as a type-level list. Every handler has a carrier of the form $\forall a.\ S_i \to M\ a$ and their composition yields a carrier $\forall a.\ (S_1, ..., S_n) \to M\ a$. Moreover, the handlers must share a common generator $(S_1, ..., S_n) \to A \to M\ B$.

Multicore OCaml comes equipped with algebraic effects and handlers intended to implement thread schedulers [7]. These handlers are similar to Eff's, but lack a type system.

The Frank language [28] shows many similarities with Eff, but allows arbitrary recursion patterns for handlers and even matching on multiple computations at the same time.

There has also been a flurry of activity in showing the relationship between row types and algebraic effects, which has seen implementations in both Links [13] and Koka [25].

8.2 Monad Transformers

Moggi [29] used monads to model side-effects while working on computational models. Independently, Spivey [38] used monads while working on a theory of exceptions. Wadler [42] popularized monads in the context of Haskell, and others [e.g., 20, 39] have sought to modularize them.

Monad transformers emerged [6, 27] from this process, and in later years various alternative implementation designs, facilitating monad (transformer) implementations, have been proposed, such as Filinksi's layered monads [9] and Jaskelioff's Monatron [17]. The Monatron library has a more structured notion of operation associated to a transformer in order to facilitate the lifting of operations [16, 18], and distinguishes algebraic operations from others.

The administrative transformations on signature coproducts (Section 4) have also been studied for monad transformers by Schrijvers and Oliveira [37].

8.3 Comparison

The only existing work that directly compares effect handlers and monads is that of Forster et al. [11]. They investigate whether the two are interexpressible. However, they do not consider modular handlers that forward unhandled operations. Also they use Filinski's layered monads rather than monad transformers for monadic effect composition and do not provide an abstraction mechanism to separate the monad's interface from its implementation.

9 Conclusion

This paper has studied the use of transformers and algebraic effects to model a variety of modular effects. Monad transformers use monad subclass constraints to allow modular syntax, and the semantics is given by a monad homomorphism. Modular algebraic effects provide signature functors whose semantics are given by folds over syntax trees. Our key contribution has been to identify modular carriers that correspond to transformers so that each approach can be given in terms of the other.

Modular effect handlers can be expressed in terms of monad transformers by working with the free monad transformer. Monad transformers for algebraic operations can be expressed in terms of effect handlers.

Finally, we have shown a limitation of the algebraic effects approach: while transformers are able to cope with scoped effects, the situation is not as clear with algebraic effects. This was first observed by Wu et al. [44], and a categorical approach to the syntax and semantics of operations with scope has since been formalised [30].

Acknowledgments

We are grateful to Koen Claessen for pointing out relevant related work, and to the anonymous reviewers for their feedback. This work has been supported by EPSRC grant number EP/S028129/1 on "Scoped Contextual Operations and Effects". Mauro Jaskelioff was supported by Agencia Nacional de Promoción Científica y Tecnológica (PICT 2016-0464). Maciej Piróg was supported by the National Science Centre, Poland under POLONEZ 3 grant "Algebraic Effects and Continuations" no. 2016/23/P/ST6/02217. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 665778.

References

[1] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

[3] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming (ICFP 2013)*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 133–144. https://doi.org/10.1145/2500365.2500581

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[5] Magnus Carlsson. 2003. Value Recursion in the Continuation Monad. (Jan. 2003). Unpublished note. http://www.carlssonia.org/ogi/mdo-callcc.pdf.

[6] Pietro Cenciarelli and Eugenio Moggi. 1993. A Syntactic Approach to Modularity in Denotational Semantics. In *CCTCS '93: Proceedings of the Conference on Category Theory and Computer Science*.

[7] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. (2015). OCaml Users and Developers Workshop, September 2015, Vancouver, Canada. http://kcsrk.info/papers/effects_ocaml15.pdf.

[8] Samuel Eilenberg and John C. Moore. 1965. Adjoint Functors and Triples. *Illinois Journal of Mathematics* 9, 3 (1965), 381–398. https://projecteuclid.org:443/euclid.ijm/1256068141

[9] Andrzej Filinski. 1999. Representing Layered Monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*, Andrew W. Appel and Alex Aiken (Eds.). 175–188. https://doi.org/10.1145/292540.292557

[10] Marcelo P. Fiore and Sam Staton. 2014. Substitution, Jumps, and Algebraic Effects. In *Annual IEEE Symposium on Logic in Computer Science (LICS 2014)*, Thomas A. Henzinger and Dale Miller (Eds.). IEEE Computer Society Press, 41:1–41:10. https://doi.org/10.1145/2603088.2603163

[11] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 13:1–13:29. https://doi.org/10.1145/3110257

[12] Jeremy Gibbons and Ralf Hinze. 2011. Just do It: Simple Monadic Equational Reasoning. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). 2–14. https://doi.org/10.1145/2034773.2034777

[13] D. Hillerström, S. Lindley, and K. Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. In *OCaml Workshop*.

[14] Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming* 9, 4 (1999), 355–372. https://doi.org/10.1017/S0956796899003500

[15] Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining Effects: Sum and Tensor. *Theoretical Computer Science* 357, 1-3 (2006), 70–99. https://doi.org/10.1016/j.tcs.2006.03.013

[16] Mauro Jaskelioff. 2009. Modular Monad Transformers. In *Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009)*, Giuseppe Castagna (Ed.). 64–79. https://doi.org/10.1007/978-3-642-00590-9_6

[17] Mauro Jaskelioff. 2011. Monatron: An Extensible Monad Transformer Library. In *Implementation and Application of Functional Languages (LNCS)*, Vol. 5836. Springer, 233–248. https://doi.org/10.1007/978-3-642-24452-0_13

[18] Mauro Jaskelioff and Eugenio Moggi. 2010. Monad Transformers as Monoid Transformers. *Theoretical Computer Science* 411, 51-52 (2010), 4441 – 4466. https://doi.org/10.1016/j.tcs.2010.09.011

[19] Mauro Jaskelioff and Russell O'Connor. 2015. A Representation Theorem for Second-Order Functionals. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/S0956796815000088

[20] 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

[21] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming (ICFP 2013)*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

[22] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell 2015)*, Ben Lippmeier (Ed.). ACM, 94–105. https://doi.org/10.1145/2804302.2804319

[23] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell 2013)*. ACM, 59–70. https://doi.org/10.1145/2503778.2503791

[24] Oleg Kiselyov and K. Sivaramakrishnan. 2016. Eff Directly in OCaml. In *OCaml Workshop*.

[25] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. http://dl.acm.org/citation.cfm?id=3009872

[26] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, 228–243. https://doi.org/10.1007/3-540-48959-2

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

[28] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 500–514. https://doi.org/10.1145/3009837.3009897

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

[30] 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)*. ACM, 809–818. https://doi.org/10.1145/3209108.3209166

[31] Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell 2014)*, Wouter Swierstra (Ed.). ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2633357.2633360

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

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

[34] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

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

[36] Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. 2015. From Monoids to Near-Semirings: The Essence of MonadPlus and Alternative. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*, Moreno Falaschi and Elvira Albert (Eds.). ACM, 196–207. https://doi.org/10.1145/2790449.2790514

[37] Tom Schrijvers and Bruno C. d. S. Oliveira. 2011. Monads, Zippers and Views: Virtualizing the Monad Stack. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP 2011)*, Olivier Danvy (Ed.). https://doi.org/10.1145/2034773.2034781

[38] M. Spivey. 1990. A Functional Theory of Exceptions. *Science of Computer Programming* 14, 1 (May 1990), 25–42. https://doi.org/10.1016/0167-6423(90)90056-J

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

[40] Wouter Swierstra. 2008. Data Types à La Carte. *Journal of Functional Programming* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758

[41] Hayo Thielecke. 1997. *Categorical Structure of Continuation Passing Style.* Ph.D. Dissertation. University of Edinburgh. Also available as technical report ECS-LFCS-97-376.

[42] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, 61–78. https://doi.org/10.1145/91556.91592

[43] Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Mathematics of Program Construction (LNCS)*, Ralf Hinze and Janis Voigtländer (Eds.), Vol. 9129. Springer, 302–322. https://doi.org/10.1007/978-3-319-19797-5

[44] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell 2014)*, Wouter Swierstra (Ed.). ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358