Monad Transformers and Modular Algebraic Effects: What Binds Them Together

Tom Schrijvers
Maciej Piróg
Nicolas Wu
Mauro Jaskelioff

Report CW 699, September 2016

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Monad Transformers and Modular Algebraic Effects: What Binds Them Together

Tom Schrijvers
Maciej Piróg
Nicolas Wu
Mauro Jaskelioff

Report CW 699, September 2016

Department of Computer Science, KU Leuven

Abstract

Monads and algebraic effects are two alternative approaches for expressing purely functional side-effects. While the two approaches have been well-studied, there is still much confusion about their relative merits and expressiveness, especially when it comes to their comparative modularity. This paper clarifies the connection between the two approaches.

In this paper we introduce the notion of *modular* algebraic effects, and show how these correspond to a specific class of monad transformers. In particular, we show that every modular algebraic effect gives rise to a monad transformer. Moreover, every monad transformer for algebraic operations gives rise to a modular effect handler. Finally, we illustrate the common ground of both approaches with an algebraic reformulation of *callCC*.

Monad Transformers and Modular Algebraic Effects: What Binds Them Together

Tom Schrijvers Maciej Piróg
KU Leuven, Belgium
tom.schrijvers@kuleuven.be
maciej.pirog@kuleuven.be

Nicolas Wu
University of Bristol, UK
nicolas.wu@bristol.ac.uk

Mauro Jaskelioff
CIFASIS-CONICET, Argentina
Universidad Nacional de Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

Monads and algebraic effects are two alternative approaches for expressing purely functional side-effects. While the two approaches have been well-studied, there is still much confusion about their relative merits and expressiveness, especially when it comes to their comparative modularity. This paper clarifies the connection between the two approaches.

In this paper we introduce the notion of *modular* algebraic effects, and show how these correspond to a specific class of monad transformers. In particular, we show that every modular algebraic effect gives rise to a monad transformer. Moreover, every monad transformer for algebraic operations gives rise to a modular effect handler. Finally, we illustrate the common ground of both approaches with an algebraic reformulation of *callCC*.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages

Keywords Handlers, Effects, Monads, Transformers, Haskell

1. Introduction

For decades monads [28, 42] have dominated the scene of functional programming with effects, and the recent popularisation of algebraic effects & handlers [4, 9, 22, 24, 34] promises to change the landscape. However, with rapid change also comes confusion, and practitioners have been left uncertain about which of the two competing approaches to choose. In this paper we show that although the approaches differ, what binds them together is greater than what drives them apart.

When one wants to obtain a combination of different effects, it is desirable to use a modular approach, in which each effect is given semantics separately. This allows for the construction of complex custom effects from simple off-the-shelf building blocks. In the monadic approach the most popular way to achieve modularity is with monad transformers [25]. Monad transformers extend an arbitrary monad to a new monad that adds an effect while at the same time ensuring that the effects provided by the original monads are available. By stacking several monad transformers in a particular order, one obtains the desired combination. In the algebraic effects approach modularity is achieved in two stages. First, the syntax of all operations involved in the desired effect is combined. Then, the syntax corresponding to each effect is given semantics by means of a handler. 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 handling unknown syntax.

Both monad transformers and modular algebraic effects address the same problem: to modularly combine purely functional definitions of explicit side-effects. Yet, it is very much an open question how the two approaches relate to each other and whether one approach is strictly superior. The differences between the two approaches makes it difficult to directly compare the two methodologies, whether in terms of their expressivity, modularity, algebraicity, or efficiency. Monad transformers have a reputation for being more expressive and efficient, while algebraic effects are said to have the upper hand when it comes to the modular composition of effects and algebraic reasoning properties. In order to make a comparison we show how, subject to certain conditions, each methodology can be expressed in terms of the other. These conditions also highlight the limitations of one approach with respect to the other. As we will see, the situation is nuanced, and it is often possible to find a middle ground where both approaches can offer their benefits.

This paper focuses only on the essential expressive power that each approach provides; other important properties, such as performance, ease of use and boilerplate automation, are beyond our scope. To study expressivity, we formulate a minimal Haskell implementation of both approaches. Similar to core calculi, these do not constitute new practical systems, but only capture the property of interest (expressivity) in existing practical systems, and abstract over all other aspects. While monad transformers are a mature technology, modular algebraic effects are in their infancy and there is no general agreement on how they should be implemented. A contribution of this paper is to provide a simple characterisation of modular handlers with which we intend to capture the essence of the available implementations. In this way, we aim to provide general insights that can be applied to everyone's favourite Haskell library.

Our specific contributions are:

1. We characterise *modular handlers* as type constructors that form a parametric family of Eilenberg-Moore algebras.
2. We compare the expressiveness of monad transformers and modular algebraic effects, 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, 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.

   These transformations are semantics-preserving.

3. We show that *callCC*, an operation that is well-known to be expressible in terms of a monad transformer, can be reformulated in terms of algebraic operations.

Paper Structure The rest of this paper is structured as follows. We start with an introduction to monad transformers and modular algebraic effects in Section 2, where we work with examples common to both to help compare and contrast the approaches. Having explained the methodologies, we show in Section 3 that it is possible to express modular algebraic effects as monad transformers. In Section 4, we then show that a restricted class of monad transformers can be turned into modular algebraic effects. In Section 5 we show how to give semantics to *callCC* in terms of algebraic operations and modular handlers, Finally, we relate our work to the rest of the field in Section 6 before concluding in Section 7.

2. Background and Overview

In this section we give an overview of monad transformers and effect handlers. We assume familiarity with monads, and introduce them here to fix notation.

2.1 Monads

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

$$join \circ return \quad = \quad id \tag{1}$$

$$join \circ fmap\ return \quad = \quad id \tag{2}$$

$$join \circ join \quad = \quad join \circ fmap\ join \tag{3}$$

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 that the order in which multiple nested contexts are collapsed is irrelevant.

In Haskell, monads are defined in terms of *return* and $(\gg\!=)$, pronounced "bind", in the following type class:

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

Valid definitions of monads must adhere to the monad laws, where

$$mx \gg\!= f \quad = \quad join\ (fmap\ f\ mx)$$

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

Using a type class restricts every type to (at most) one monadic interpretation, the one given by the *Monad* class. For instance, state can be encapsulated by the following type:

```
newtype State s a = State {runState :: s → (a, s)}
```

This is a monad under the following type class instance:

```
instance Monad (State s) where
  return x       = State (λs → (x, s))
  State p ≫= k = State (λs → let (x, s′) = p s in
                                   runState (k x) s′)
```

This monad facilitates the use of a state of type *s* that is threaded around from one monadic computation to the next, where it might be accessed or replaced with a different one. There are two associated functions, *get* and *put*:

```
get :: State s s
get = State (λs → (s, s))
put :: s → State s ()
put s = State (λ_ → ((), s))
```

The *get* operation creates a computation that returns the state that it is given, leaving the state unchanged, while *put s* creates a computation that ignores the state that it is given and changes the state to *s*.

Monads often come with *run* functions that are responsible for extracting values from monadic computations. One example is the function $runState :: State\ s\ a \to s \to (s, a)$ defined above. For the sake of abstraction, we prefer not to work directly with implementations of monads to extract the final results, but we rely on such functions, even though some of them, like *runState*, simply reveal the implementation.

2.2 Monad Transformers

Monad transformers allow monads to be extended with additional functionality. This is done by providing a function that lifts one monad into another.

Lifting The *Trans* type class is the well-known Haskell interface for monad transformers.

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

This interface states that a transformer *t* provides a *lift* function that lifts a computation in any monad *m* to a corresponding computation in the transformed monad *t m*. An implicit requirement for instances of this class is that whenever *m* is a monad, *t m* should be a monad. Moreover, *lift* should be a monad (homo)-morphism, where it respects the following properties:

$$lift \circ return \quad = \quad return \tag{4}$$

$$lift \circ join \quad = \quad join \circ lift \circ fmap\ lift \tag{5}$$

This states 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 → m (a, s)}
```

A computation is lifted into $State_T\ s$ as follows:

```
instance Trans (State_T s) where
  lift m = State_T (λs → m ≫= λa → return (a, s))
```

We must also provide an instance *Monad* (*t m*) when we have *Monad m* and *Trans t*. For $State_T\ s$, this gives us:

```
instance Monad m ⇒ Monad (State_T s m) where
  return x         = State_T (λs → return (x, s))
  State_T p ≫= k = State_T (λs → do (x, s′) ← p s
                                         runState_T (k x) s′)
```

This definition is almost identical to its counterpart for *State*, except that we are now careful to thread the monadic effects of *m* throughout the computation.

Operations The *Trans* type class provides only a part of a transformer's interface. We also expect functions that allow access to the interesting aspects of the monadic behaviour that has been added. Although it is possible to give these functions in terms of a concrete instance, it is better practice to provide a type class that encapsulates the key operations, along with some laws that pin down their semantics.

In the case of $State_T\ s$, we should anticipate functions similar to *get* and *put* from *State s*. Since we do not want to rely on their implementation, we can instead consider the properties that these operations satisfy [12].

$$\begin{aligned} get \gg\!\!= put &= id \\ put\ s \gg put\ s' &= put\ s' \\ get \gg get &= get \\ put\ s \gg get &= put\ s \gg return\ s \end{aligned}$$

Thinking about operations in terms of properties rather than implementations calls for us to incorporate *get* and *put* into a typeclass that embodies stateful computations.

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

Legal instances of this class adhere to the laws above. Without them, there is no way to arbitrate between different implementations.

With this in place, we can give an instance not only for *State s*, using the definitions we introduced before, but also for $State_T\ s\ m$:

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

These definitions are very similar to the counterparts for *State s*, differing only by the fact that pure values have now been wrapped within a monadic context.

We can now write programs that leave the type they operate on abstract, allowing programs like *incr* to work both for *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, for a monadic *X*, the operations added by a transformer are captured in a so-called monad subclass *MonadX*.

```
class Monad m ⇒ MonadX m where
  op1 :: ... → m T1
  ...
  opn :: ... → m Tn
```

Again, we would expect there to be laws associated to this class.

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

The transformer approach requires there to be a monad at the bottom of the stack of transformers. It is convenient to use the identity monad *Id* for this purpose.

```
newtype Id a = Id {runId :: a}
instance Monad Id where
  return = Id
  Id x ≫= f = f x
```

The bind operation for this monad is essentially function application.

As an alternative to giving the type of a computation directly in terms of the composition of transformers, we can instead work with class constraints: a computation with the type $(MonadX_1\ m, MonadX_2\ m) \Rightarrow m\ a$ could be satisfied by either $T_1\ (T_2\ Id)$, or $T_2\ (T_1\ Id)$, thus allowing us to interpret one fragment of code in whichever way we choose.

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

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

Here we expect only one law to hold, which expresses that no computation is performed after failure:

$$fail \gg\!\!= f \quad = \quad fail$$

The law is satisfied by the familiar *Maybe* type:

```
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 clearly 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 MaybeT m a = MaybeT {runMaybeT :: m (Maybe a)}
instance Monad m ⇒ Monad (MaybeT m) where
  return x = MaybeT (return (Just x))
  MaybeT mmx ≫= f = MaybeT (do
    mx ← mmx
    case mx of
      Nothing → return Nothing
      Just x  → runMaybeT (f x))
instance Monad m ⇒ MonadFail (MaybeT m) where
  fail = MaybeT (return Nothing)
instance Trans MaybeT where
  lift mx = MaybeT (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*, we can express computations that make use of the two effects. Consider the following example:

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

By keeping the type of this computation abstract, we are free to choose the exact semantics at the point of application. For instance, we might want to evaluate *prog* as a computation of type *Maybe* ((), *Int*) that returns the state only when there are no exceptions, and *Nothing* otherwise. To do so, we must show how a monad *m* that supports failure can be promoted through a $State_T$ transformer:

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

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

```
> (runId ∘ runMaybeT ∘ flip runStateT 0) prog
Nothing
```

Since *prog* fails during the computation no state is returned, and we are left only with the fact that an error occurred.

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 occurrence of the *fail*.

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 $run :: Monad\ m \Rightarrow T\ m\ A \to m\ B$.

The two 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 in which a given operation can be lifted, and the choice is dependent on the expected semantics. Therefore, one usually needs to manually write one instance that implements the lifting of operations for each monad transformer combination (as was done above).

2.3 Algebraic Effects

The algebraic-effect approach is organized in a quite different way. It distinguishes between the syntax and the semantics of effects.

Syntax The syntax of an effect consists of the signatures of the supported operations. These come in the form of a functor *sig*. For the state effect we have the functor STATE *s*:

```
data STATE s k = GET (s → k) | PUT s (() → k)
```

The two constructors GET and PUT denote operations for reading and writing the state, respectively. Each constructor has a field that holds the continuation of the operation, $s \to k$ for GET and $() \to k$ for PUT. Here, the type parameter *k* is a place-holder for the type of computations. It should only be used in the position of continuations.

The *free monad* is the recursive structure that sequentially composes zero or more operations of the given signature. Its basic definition in Haskell is:

```
data Free sig a = Return a | Op (sig (Free sig a))
```

Here, *Return* expresses a trivial computation over a signature *sig* that performs no operation and immediately returns a value of type *a*. In contrast, *Op* denotes a computation that performs one operation whose continuation consists again of zero or more operations. One can think of this construction as a syntax tree whose node shape is determined by *sig*. The result of *return x* is a leaf, and $(\gg=)$ grows the syntax tree of operations at its leaves.

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

Putting syntax together using these constructs directly can be a little cumbersome. For instance, reproducing the program *incr* in this style would require the following code:

```
incr′ :: Free (STATE Int) ()
incr′ = Op (GET (λs → Op (PUT (s + 1) Return)))
```

This can be somewhat alleviated by creating smart constructors, where *return* has been used for the continuation parameter.

```
get′ :: Free (STATE s) s
get′ = Op (GET return)
put′ :: s → Free (STATE s) ()
put′ s = Op (PUT s return)
```

Semantics The most straightforward way to interpret—or *handle*—the *Free f* syntax of effects is by means of a *fold* over the structure.

```
fold :: Functor sig ⇒ (a → b) → (sig b → b) → (Free sig a → b)
fold gen alg (Return 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*.

For instance, $\langle s \to a, gen_S, alg_S\rangle$ is a handler for *Free* (STATE *s*) *a* terms:

```
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
```

2.4 Modular Algebraic Effects

The modular composition of effects does not follow automatically from the algebraic effects approach. We need to impose additional structure that allows us to compose both signatures and handlers.

Modular Signatures Signatures compose naturally with the functor coproduct:

```
data (sig_1 + sig_2) a = Inl (sig_1 a) | Inr (sig_2 a)
```

The VOID functor is the neutral element of this functor coproduct, and is expressed as a type with no constructors.

```
data VOID k
```

Any signature *sig* is isomorphic to *sig* + VOID. This means that VOID serves as a nice base case with the trivial handler:

```
runVOID :: Free VOID a → a
runVOID = fold gen_V alg_V where
  gen_V :: a → a
  gen_V = id
  alg_V :: VOID a → a
  alg_V = ⊥
```

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:

```
(▽) :: (sig_1 a → b) → (sig_2 a → b) → ((sig_1 + sig_2) a → b)
(alg_1 ▽ alg_2) (Inl op) = alg_1 op
(alg_1 ▽ alg_2) (Inr op) = alg_2 op
```

For algebras it is always the case that $a = b$. Depending on the operation that is present, the appropriate algebra is applied.

However, this coincidence of carrier type and generator is rather exceptional, and in this paper we focus on a different composition scheme that occurs more commonly in the literature. The idea is to run one handler after the other, first interpreting $\textsc{Sig}_1$ without touching $\textsc{Sig}_2$ and then interpreting the latter.

$$\begin{array}{ll}
\quad \mathit{Free}\ (\textsc{Sig}_1 + \textsc{Sig}_2 + \textsc{Void})\ A_1 & \{\ \mathit{fold}\ \mathit{gen}_1\ \mathit{alg}_1\ \} \\
\to H_1 & \{\ \mathit{run}_1\ \} \\
\to \mathit{Free}\ (\textsc{Sig}_2 + \textsc{Void})\ A_2 & \{\ \mathit{fold}\ \mathit{gen}_2\ \mathit{alg}_2\ \} \\
\to H_2 & \{\ \mathit{run}_2\ \} \\
\to \mathit{Free}\ \textsc{Void}\ A_3 & \{\ \mathit{run}\textsc{Void}\ \} \\
\to A_3 &
\end{array}$$

In this scheme, run_1 denotes a run function that mediates between the carrier type H_1 of the first *fold* and the new computation type $\mathit{Free}\ (\textsc{Sig}_2 + \mathit{Void})\ A_2$ that is processed by the next handler. In trivial cases, we have that $H_1 \equiv \mathit{Free}\ (\textsc{Sig}_2 + \mathit{Void})\ A_2$ and $\mathit{run}_1 \equiv \mathit{id}$ need not be mentioned explicitly. Another common case is $H_1 \equiv S \to \mathit{Free}\ (\textsc{Sig}_2 + \mathit{Void})\ A_2$ where, e.g., S is a type of state; in this case there is a whole family of run functions of the form $\mathit{run}_1 \equiv \lambda p \to p\ s$, one for each possible initial state $s :: S$.

The carrier types H_1 and H_2 in the above scheme are not entirely satisfactory: they do not capture the modularity of effects. By modularity of effects we mean two specific properties. Firstly, an effect's handler should work regardless of the other effects present. Secondly, the effect's handler should leave the syntax of other effects alone when it handles programs.

We capture these requirements with carrier types $H_1 :: (* \to *) \to *$ that are parametric in the remaining effects. Moreover, we abstract those remaining effects to a polymorphic monad type parameter m that does not reveal the free monadic structure and thus does not allow any of the operations to be observed or duplicated. Taken together, this results in the following signatures for a modular handler's algebra and generator:

$$\begin{array}{l}
\mathit{alg} :: \mathit{Monad}\ m \Rightarrow \textsc{Sig}_1\ (H_1\ m) \to H_1\ m \\
\mathit{gen} :: \mathit{Monad}\ m \Rightarrow A_1 \to H_1\ m
\end{array}$$

Note that the above algebra only takes care of $\textsc{Sig}_1$ operations. In order to deal with the operations of other effects, we require an additional algebra for the remaining $\textsc{Sig}_2$ effects that is also parametric in those effects, i.e., it should treat them uniformly. We capture this additional algebra in a type class for modular carriers:

$$\begin{array}{l}
\mathbf{class}\ \mathit{ModularCarrier}\ c\ \mathbf{where} \\
\quad \mathit{joinl} :: \mathit{Monad}\ m \Rightarrow m\ (c\ m) \to c\ m
\end{array}$$

where *joinl* is a parametric family of Eilenberg-Moore algebras, i.e., it respects the monad operations:

$$\mathit{joinl} \circ \mathit{return} \quad = \quad \mathit{id} \tag{6}$$

$$\mathit{joinl} \circ \mathit{join} \quad = \quad \mathit{joinl} \circ \mathit{fmap}\ \mathit{joinl} \tag{7}$$

These laws express an asymmetric version of the monad laws, where only a left-sided join is provided. With such a modular carrier H_1 of a given $\textsc{Sig}_1$-algebra alg_1, we can then derive a $(\textsc{Sig}_1 + \textsc{Sig}_2)$-algebra that leaves the $\textsc{Sig}_2$ syntax alone.

$$\begin{array}{l}
\mathit{liftAlg} :: \forall \mathit{sig}_1\ \mathit{sig}_2\ c_1 . (\mathit{Functor}\ \mathit{sig}_2, \mathit{ModularCarrier}\ c_1) \\
\qquad \Rightarrow (\qquad\quad \mathit{sig}_1\ \ (c_1\ (\mathit{Free}\ \mathit{sig}_2)) \to (c_1\ (\mathit{Free}\ \mathit{sig}_2))) \\
\qquad \to ((\mathit{sig}_1 + \mathit{sig}_2)\ (c_1\ (\mathit{Free}\ \mathit{sig}_2)) \to (c_1\ (\mathit{Free}\ \mathit{sig}_2))) \\
\mathit{liftAlg}\ \mathit{alg}_1 = \mathit{alg}_1 \triangledown \mathit{alg}_2\ \mathbf{where} \\
\quad \mathit{alg}_2 :: \mathit{sig}_2\ (c_1\ (\mathit{Free}\ \mathit{sig}_2)) \to c_1\ (\mathit{Free}\ \mathit{sig}_2) \\
\quad \mathit{alg}_2\ \mathit{op} = \mathit{joinl}\ (\mathit{Op}\ (\mathit{fmap}\ \mathit{return}\ \mathit{op}))
\end{array}$$

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

Example To illustrate the above, we can give a modular handler for state, which is similar to the handler for $\mathit{Free}\ (\textsc{State}\ s)\ a$ discussed earlier, except that the result a appears in a monadic context m. Since we want to provide an instance for the modular carrier type, we will wrap it in a newtype constructor:

$$\mathbf{newtype}\ \mathit{State}_H\ s\ a\ m = \mathit{State}_H\ \{\mathit{runState}_H :: s \to m\ a\}$$

The *ModularCarrier* instance for this type is:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{ModularCarrier}\ (\mathit{State}_H\ s\ a)\ \mathbf{where} \\
\quad \mathit{joinl}\ \mathit{mf} = \mathit{State}_H\ (\lambda s \to \mathbf{do}\ f \leftarrow \mathit{mf}; \mathit{runState}_H\ f\ s)
\end{array}$$

Now the semantics of $\textsc{State}$ can be given as follows:

$$\begin{array}{l}
\mathit{gen}_{SH} :: \mathit{Monad}\ m \Rightarrow a \to \mathit{State}_H\ s\ a\ m \\
\mathit{gen}_{SH}\ x = \mathit{State}_H\ (\lambda s \to \mathit{return}\ x)
\end{array}$$

$$\begin{array}{l}
\mathit{alg}_{SH} :: \mathit{Monad}\ m \Rightarrow \textsc{State}\ s\ (\mathit{State}_H\ s\ a\ m) \to \mathit{State}_H\ s\ a\ m \\
\mathit{alg}_{SH}\ (\textsc{Get}\ k) \quad\ = \mathit{State}_H\ (\lambda s\ \to \mathit{runState}_H\ (k\ s)\ s) \\
\mathit{alg}_{SH}\ (\textsc{Put}\ s\ k) = \mathit{State}_H\ (\lambda\_ \to \mathit{runState}_H\ (k\ ())\ s)
\end{array}$$

With an appropriate run function we get the modular $\textsc{State}\ \mathit{Int}$ handler $\langle \mathit{State}_H\ \mathit{Int}\ \mathit{Int}, \mathit{alg}_{SH}, \mathit{gen}_{SH}, \mathit{flip}\ \mathit{runState}_H\ 5 \rangle$:

$$\begin{array}{l}
\mathbf{>}\ (\mathit{runId} \circ \mathit{flip}\ \mathit{runState}_H\ 5 \circ \mathit{fold}\ \mathit{gen}_{SH}\ \mathit{alg}_{SH})\ (\mathit{incr}' \gg \mathit{get}') \\
6
\end{array}$$

Since $\mathit{State}_H\ s\ a$ is a modular carrier we can compose it with other effects. As a second effect to compose with, we will again show how failure can be modelled, this time using effect handlers.

$$\mathbf{data}\ \textsc{Fail}\ k = \textsc{Fail}$$

$$\mathit{fail}' = \mathit{Op}\ \textsc{Fail}$$

One handler for this is given by $\langle \mathit{Maybe}_H\ a\ m, \mathit{gen}_F, \mathit{alg}_F \rangle$, where the modular carrier is defined as:

$$\mathbf{newtype}\ \mathit{Maybe}_H\ a\ m = \mathit{Maybe}_H\ \{\mathit{runMaybe}_H :: m\ (\mathit{Maybe}\ a)\}$$

$$\begin{array}{l}
\mathbf{instance}\ \mathit{ModularCarrier}\ (\mathit{Maybe}_H\ a)\ \mathbf{where} \\
\quad \mathit{joinl}\ \mathit{mf} = \mathit{Maybe}_H\ (\mathbf{do}\ f \leftarrow \mathit{mf}; \mathit{runMaybe}_H\ f)
\end{array}$$

Observe that Maybe_H uses the same representation as Maybe_T, but has its type parameters swapped; in Section 4.2 we show that other monad transformers give rise to modular carriers in a similar manner.

The associated generator and algebra are:

$$\begin{array}{l}
\mathit{gen}_F :: \mathit{Monad}\ m \Rightarrow a \to \mathit{Maybe}_H\ a\ m \\
\mathit{gen}_F\ x = \mathit{Maybe}_H\ (\mathit{return}\ (\mathit{Just}\ x))
\end{array}$$

$$\begin{array}{l}
\mathit{alg}_F :: \mathit{Monad}\ m \Rightarrow \textsc{Fail}\ (\mathit{Maybe}_H\ a\ m) \to \mathit{Maybe}_H\ a\ m \\
\mathit{alg}_F\ \textsc{Fail} = \mathit{Maybe}_H\ (\mathit{return}\ \mathit{Nothing})
\end{array}$$

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

$$\begin{array}{l}
\mathit{hdl}_{FS} :: s \to \mathit{Free}\ (\textsc{Fail} + (\textsc{State}\ s + \textsc{Void}))\ a \to \mathit{Maybe}\ a \\
\mathit{hdl}_{FS}\ s = \mathit{run}\textsc{Void} \circ \\
\qquad\qquad \mathit{flip}\ \mathit{runState}_H\ s \circ \mathit{fold}\ \mathit{gen}_{SH}\ (\mathit{liftAlg}\ \mathit{alg}_{SH}) \circ \\
\qquad\qquad \mathit{runMaybe}_H \circ \mathit{fold}\ \mathit{gen}_F\ (\mathit{liftAlg}\ \mathit{alg}_F)
\end{array}$$

$$\begin{array}{l}
\mathit{hdl}_{SF} :: s \to \mathit{Free}\ (\textsc{State}\ s + (\textsc{Fail} + \textsc{Void}))\ a \to \mathit{Maybe}\ a \\
\mathit{hdl}_{SF}\ s = \mathit{run}\textsc{Void} \circ
\end{array}$$

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

$$\begin{array}{l}
\quad \mathit{runMaybe}_H \circ \mathit{fold}\ \mathit{gen}_F\ (\mathit{liftAlg}\ \mathit{alg}_F) \circ \\
\quad \mathit{flip}\ \mathit{runState}_H\ s \circ \mathit{fold}\ \mathit{gen}_{SH}\ (\mathit{liftAlg}\ \mathit{alg}_{SH})
\end{array}$$

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 $\langle \mathit{Def}\ m, \mathit{gen}_D, \mathit{alg}_D, \mathit{runDef} \rangle$ for failure that only works for *Int* computations.

$$\begin{array}{l}
\textbf{newtype}\ \mathit{Def}\ m = \mathit{Def}\ \{ \mathit{runDef} :: m\ \mathit{Int} \} \\
\textbf{instance}\ \mathit{ModularCarrier}\ \mathit{Def}\ \textbf{where} \\
\quad \mathit{joinl}\ \mathit{mf} = \mathit{Def}\ (\textbf{do}\ f \leftarrow \mathit{mf}; \mathit{runDef}\ f) \\
\mathit{gen}_D :: \mathit{Monad}\ m \Rightarrow \mathit{Int} \rightarrow \mathit{Def}\ m \\
\mathit{gen}_D\ x = \mathit{Def}\ (\mathit{return}\ x) \\
\mathit{alg}_D :: \mathit{Monad}\ m \Rightarrow \mathrm{F{\scriptsize AIL}}\ (\mathit{Def}\ m) \rightarrow \mathit{Def}\ m \\
\mathit{alg}_D\ \mathrm{F{\scriptsize AIL}} = \mathit{Def}\ (\mathit{return}\ 0)
\end{array}$$

This modular handler does not abort the computation upon failure, but instead proceeds with the default value 0. Section 3 shows how to formulate similar monad transformers that can only be run with computations of a particular value type.

2.5 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

$$\mathit{MonadState}\ s\ m \Rightarrow m$$

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

$$(\mathit{MonadState}\ s\ m, \mathit{MonadFail}\ m) \Rightarrow m$$

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

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

$$\mathit{Free}\ (\mathrm{S{\scriptsize TATE}}\ s + \mathrm{F{\scriptsize AIL}})$$

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

$$\mathit{Functor}\ \mathit{sig} \Rightarrow \mathit{Free}\ (\mathrm{S{\scriptsize TATE}}\ s + \mathit{sig}).$$

Because the monad transformer approach uses type-class constraints in order to overload syntax, the order in which they are written does not matter. For example, $(\mathit{MonadState}\ s\ m, \mathit{MonadFail}\ m)$ is the same constraint as $(\mathit{MonadFail}\ m, \mathit{MonadState}\ s\ m)$. 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 [24, 39].

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 an operation of a monad to the transformer monad. In the algebraic effects approach, effects are introduced with algebraic operations as determined by a signature. Algebraic operations for a signature functor *SIG* correspond to functions $\forall a. \mathit{SIG}\ a \rightarrow M\ a$. 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 [31] 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 in the monad transformer approach one would require a special lifting function. Therefore, there seems to be a trade-off: either we provide liftings for scoped operations such as *catch*, or we lose some modularity.

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 monad transformers that satisfies all the type class constraints. The variation in semantics is possible because there are different monad transformer stacks that satisfy the same set of constraints.

Firstly, we can order the same transformers in different ways, which gives rise to different interactions between the effects. For instance, both $\mathit{State}_T\ s\ (\mathit{Maybe}_T\ \mathit{Id})$ and $\mathit{Maybe}_T\ (\mathit{State}_T\ s\ \mathit{Id})$ satisfy the constraints $(\mathit{MonadState}\ s\ m, \mathit{MonadFail}\ m)$, but give rise to different interactions between state and failure handling.

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

$$\begin{array}{l}
\textbf{newtype}\ \mathit{LogState}_T\ s\ m\ a = \mathit{LST}\ \{ \mathit{runLST} :: s \rightarrow m\ (a, s, [s]) \} \\
\textbf{instance}\ \mathit{Trans}\ (\mathit{LogState}_T\ s)\ \textbf{where} \\
\quad \mathit{lift}\ m = \mathit{LST}\ (\lambda s \rightarrow \textbf{do}\ x \leftarrow m; \mathit{return}\ (x, s, [\,])) \\
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{LogState}_T\ s\ m)\ \textbf{where} \\
\quad \mathit{return}\ x = \mathit{LST}\ (\lambda s \rightarrow \mathit{return}\ (x, s, [\,])) \\
\quad m \gg\!\!= f\ = \mathit{LST}\ (\lambda s \rightarrow \textbf{do}\ (x, s', h_1)\ \leftarrow \mathit{runLST}\ m\ s \\
\qquad\qquad\qquad\qquad\qquad\quad (y, s'', h_2) \leftarrow \mathit{runLST}\ (f\ x)\ s' \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (y, s'', h_1 +\!\!+ h_2)) \\
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ (\mathit{LogState}_T\ s\ m)\ \textbf{where} \\
\quad \mathit{get} \quad = \mathit{LST}\ (\lambda s \rightarrow \mathit{return}\ (s, s, [\,])) \\
\quad \mathit{put}\ s = \mathit{LST}\ (\lambda s' \rightarrow \mathit{return}\ ((), s, [s']))
\end{array}$$

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:

$$\begin{array}{l}
\textbf{newtype}\ \mathit{LogStateH}\ s\ a\ m = \mathit{LS}\ \{ \mathit{runLSH} :: s \rightarrow [s] \rightarrow m\ (a, [s]) \} \\
\mathit{gen}_{LS} :: \mathit{Monad}\ m \Rightarrow a \rightarrow \mathit{LogStateH}\ s\ a\ m \\
\mathit{gen}_{LS}\ x = \mathit{LS}\ (\lambda s\ h \rightarrow \mathit{return}\ (x, \mathit{reverse}\ h)) \\
\mathit{alg}_{LS} :: \mathrm{S{\scriptsize TATE}}\ s\ (\mathit{LogStateH}\ s\ a\ m) \rightarrow \mathit{LogStateH}\ s\ a\ m \\
\mathit{alg}_{LS}\ (\mathrm{G{\scriptsize ET}}\ k) \quad = \mathit{LS}\ (\lambda s\ \ h \rightarrow \mathit{runLSH}\ (k\ s)\ \ s\ h) \\
\mathit{alg}_{LS}\ (\mathrm{P{\scriptsize UT}}\ s\ k) = \mathit{LS}\ (\lambda s'\ h \rightarrow \mathit{runLSH}\ (k\ ())\ s\ (s' : h))
\end{array}$$

Similar to the transformer approach, we may also control the effect interaction by running the handlers in different orders[2]. 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 partially determined by the structure of 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 will 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.

3. 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.

3.1 From Signature to Monad Subclass

Whereas in the algebraic effects approach operations are characterised by a signature functor, in the monad transformer approach operations are given by a monad subclass. We define a monad subclass in terms of a given signature functor *sig*.

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

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.

Instances of *MonadEff sig m* are in a one-to-one correspondence with operations $op :: sig\ (m\ a) \to m\ a$ with the following property:

$$join \circ op = op \circ fmap\ join \tag{8}$$

In fact, this is how algebraic operations are usually presented. However, we prefer our implementation, in which algebraicity is enforced by the type system alone.

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

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

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

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

One instance of this class is the following, which shows that for any signature *sig*, the effect of interpreting it in the free monad is:

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

This places a *return* at every continuation.

Alternative Signatures The type class *MonadEff* provides a generic operation which is not always the most convenient form for operations. To counter this, we can usually provide more convenient auxiliary operations, by using so-called generic effects [30]. Consider the following (parametrised) functor:

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

For this functor we have the following equivalence [19]. For all functors *m*, and types *a* and *b*,

$$a \to m\ b \quad\cong\quad \forall x.\, \textit{SIG}\ a\ b\ x \to m\ x \tag{9}$$

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.

For example, for the state effect, we can recover the *MonadState* interface of Section 2.2. The signature *STATE s* is isomorphic to *SIG* () *s* + *SIG s* (). By equivalence (9) we obtain 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)
```

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.

3.2 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 ⟨*H*, *gen*, *alg*, *run*⟩. In fact, we have two ways to do so.

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 $Free_T\ sig_1\ m$ implements both the operations from the signature sig_1 and the algebraic operations provided by the monad *m*. Thus, we obtain the following instance:

```
instance (Functor sig1, MonadEff sig2 m)
  ⇒ MonadEff (sig1 + sig2) (FreeT sig1 m) where
  eff (Inl op) = FreeT (return (OpF (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.

[2] In our crude implementation, we would also need to re-arrange the order of the coproduct of signatures to match the order of handlers.

```
foldFree_T :: (Monad m, Functor sig, ModularCarrier h)
           ⇒ (a → h m)
           → (sig (h m) → h m)
           → Free_T sig m a → h m
foldFree_T gen alg = goM where
  goM = joinl ∘ 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 sig_1, Functor sig_2)
         ⇒ Free (sig_1 + sig_2) a → Free_T sig_1 (Free sig_2) a
toFree_T = fold return (join ∘ eff)
fromFree_T :: (Functor sig_1, Functor sig_2)
           ⇒ Free_T sig_1 (Free sig_2) a → Free (sig_1 + sig_2) 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 surely recognise this as an obvious consequence of Hyland, Plotkin, and Power's [15] characterisation of *Free sig m* as a coproduct of *m* and *Free sig* in the category of monads and monad morphisms.) Moreover, the respective folds correspond:

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

The involved types and functions between them are summarised in the commutative diagram in Figure 1(a).

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

```
runFree_T :: (Monad m, Functor sig, ModularCarrier h)
          ⇒ (a → h m) → (sig (h m) → h m) → (h m → m b)
          → Free_T sig m a → m b
runFree_T gen alg run = run ∘ 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 directly captures in its carrier type the intended denotation *h*. We can obtain this with a particular instance of the continuation monad.

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

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

```
newtype ContH h (m :: ∗ → ∗) a =
  ContH {unContH :: (a → h m) → h m}
instance Monad (ContH r m) where
  return x = ContH (λk → k x)
  m ≫= k = ContH $ λc → unContH m (λx → unContH (k x) c)
```

When *h* is a modular carrier, *ContH h* is a monad transformer.

```
instance ModularCarrier h ⇒ Trans (ContH h) where
  lift m = ContH (λk → joinl (fmap k m))
```

For any fixed signature *SigH*, modular carrier *H*, and algebra $algH :: SigH\ (H\ m) \to H\ m$, we may define *ContH H m* to be a signature monad, using the following generic function *effContH*:

```
effContH :: (Functor sig)
         ⇒ (sig (h m) → h m)
         → sig a → ContH h m a
effContH alg s = ContH (λk → alg (fmap k s))
instance (MonadEff sig_2 m)
  ⇒ MonadEff (SigH + sig_2) (ContH H m) where
    eff (Inl op) = effContH algH op
    eff (Inr op) = lift (eff op)
```

We embed syntax in the *ContH h* transformer using the function *toContH alg*. Unlike the transformation into the free monad transformer, the application of *toContH alg* is effectively giving semantics to the syntax, and therefore there is no way back.

```
toContH :: (Functor sig_1, Functor sig_2, ModularCarrier h)
        ⇒ (∀m. Monad m ⇒ sig_1 (h m) → h m)
        → Free (sig_1 + sig_2) a → ContH h (Free sig_2) a
toContH alg = fold return (join ∘ (effContH alg ∇ (lift ∘ eff)))
```

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

$$fold\ gen\ (liftAlg\ alg) = unContH\ gen \circ toContH\ alg$$

The involved types, functions, and equations are summarised in the commutative diagram in Figure 1(b). Again, the run function puts everything together:

```
runContH :: (Monad m, Functor sig, ModularCarrier h)
         ⇒ (a → h m) → (h m → m b)
         → ContH h m a → m b
runContH gen run = run ∘ unContH gen
```

We have shown that it is possible to embed the algebraic-operations approach into the monad-transformer approach in two ways. The first embedding goes through the free monad transformer. This transformer is well known and there are techniques that lead to efficient implementations [20, 41]. On the other hand, fusion of computations is more easily achieved for the *ContH* transformer of the second embedding [43].

4. 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 of the members of 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 3.1.

4.1 Transformer Signature

As discussed in Section 3.1, algebraic operations come in many different guises. However, in the monad-transformer approach, they usually are 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 (9), we obtain that the signature functor for such an operation must be *Sɪɢ A B*.

Any type variables that are quantified at the level of the type class *MonadX* become type parameters of the signature functor. This happens for instance, in the case of the *s* type parameter of *MonadState*. The methods *get* and *put* of the *MonadState* class can be written in the above form as:

```
get :: () → m s
put :: s  → m ()
```

(a) The free monad transformer:

$$\begin{tikzcd}
sig_1\ a \arrow[d, "eff \circ Inl"'] \arrow[rr, "eff \circ Inl"] & & sig_2\ a \arrow[d, "eff \circ Inr"] \arrow[ll, "eff \circ Inr"'] \\
Free\ (sig_1 + sig_2)\ a \arrow[rr, shift left, "toFree_T"] \arrow[dr, "fold\ gen\ (liftAlg\ alg)"'] & & Free_T\ sig_1\ (Free\ sig_2)\ a \arrow[ll, shift left, "fromFree_T"] \arrow[dl, "foldFree_T\ gen\ alg"] \\
& h\ (Free\ sig_2) &
\end{tikzcd}$$

(b) The continuation monad:

$$\begin{tikzcd}
sig_1\ a \arrow[d, "eff \circ Inl"'] \arrow[rr, "effContH\ alg"] & & sig_2\ a \arrow[d, "lift \circ eff"] \arrow[ll, "eff \circ Inr"'] \\
Free\ (sig_1 + sig_2)\ a \arrow[rr, "toContH\ alg"] \arrow[dr, "fold\ gen\ (liftAlg\ alg)"'] & & ContH\ h\ (Free\ sig_2)\ a \arrow[dl, "unContH\ gen"] \\
& h\ (Free\ sig_2) &
\end{tikzcd}$$

Figure 1. Summary of monad transformers induced by handlers

where the trivial unit parameter of *get* makes the operation type fit. This yields the signature functor:

data $\textit{State}'\ s\ k = \textit{Get}'\ ()\ (s \to k) \mid \textit{Put}'\ s\ (() \to k)$

Since the unit type () is trivial, we can remove it from the signature, thus giving us the signature *STATE* discussed in Section 2.3.

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 the *MonadPlus* typeclass.

```
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 on *a* means 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 unit (as we did for *get*), we arrive at the following equivalent type for *mzero*:

$mzero :: () \to m\ Void$

The type of the operation *mzero* gives rise to the signature functor:

data $\textit{MPlus}\ k = \textit{MZero}\ ()\ (Void \to k) \mid \ldots$

Since both the unit type () and the type $(Void \to r)$ are trivial, we can remove them.

data $\textit{MPlus}\ k = \textit{MZero} \mid \ldots$

But how can we make *mplus* fit?

Signature Reformulation Sometimes, the traditional formulation of a method signature does not fit the pattern, but the signature can be reformulated into an equivalent form.

Although the operation $mplus :: m\ a \to m\ a \to m\ a$ does not fit the required shape, we can restrict ourselves to those instances that satisfy property (8).[3] With this assumption, the operation *mplus* is algebraic and is equivalent to an operation

$choose :: a \to a \to m\ a$

for non-deterministic choice between two values. The operation *choose* is in the right shape of an algebraic operation for the functor *CHOOSE*:

data $\textit{Choose}\ k = \textit{Choose}\ k\ k$

Incorporating this into *MPLUS* by renaming the constructor, we arrive at the following signature for the *MonadPlus* typeclass:

data $\textit{MPlus}\ k = \textit{MZero} \mid \textit{MPlus}\ k\ k$

This is, of course, isomorphic to $\textit{Fail} + \textit{Choose}$.

This is a perfectly standard signature, which we can also give a semantics with the handler $\langle MPlus_H\ m\ a, gen_{MP}, alg_{MP} \rangle$:

```
newtype MPlus_H a m = MPlus_H {runMPlus_H :: m [a]}
instance ModularCarrier (MPlus_H a) where
  joinl = MPlus_H ∘ join ∘ fmap runMPlus_H

gen_MP :: Monad m ⇒ a → MPlus_H a m
gen_MP x = MPlus_H (return [x])

alg_MP :: Monad m ⇒ MPLUS (MPlus_H a m) → MPlus_H a m
alg_MP (MPLUS k1 k2) = MPlus_H (do xs ← runMPlus_H k1
                                   ys ← runMPlus_H k2
                                   return (xs ++ ys))
alg_MP (MZERO)       = MPlus_H (return [ ])
```

Since *joinl* is isomorphic to *join*, $MPlus_H\ a$ is clearly a modular carrier.

4.2 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 WrapT a m = WrapT {unWrapT :: T m a}
```

The monadic structure readily provides implementations for the operations and for the generator. In detail, if *T* implements a signature $\textit{Sig}_T$ via a function $eff_T :: Monad\ m \Rightarrow \textit{Sig}_T\ a \to T\ m\ a$. Then, we define the algebra and the generator respectively as:

```
alg_T :: (Monad m) ⇒ SIG_T (WrapT a m) → WrapT a m
alg_T = WrapT ∘ join ∘ eff_T ∘ fmap unWrapT

gen_T :: Monad m ⇒ a → WrapT a m
gen_T = WrapT ∘ return
```

The transformer *T* also makes for a modular carrier:

```
instance ModularCarrier (WrapT a) where
  joinl = WrapT ∘ join ∘ lift ∘ fmap unWrapT
```

With this, we define the following handler:

```
hdlT :: Free (SIG_T + sig) a → WrapT a (Free sig)
hdlT = WrapT ∘ fold gen_T (liftAlg alg_T)
```

The correctness of this embedding is in the fact that *unWrapT hdlT* is a monad morphism, and that it respects the eff_T function in the sense that

$$eff_T \triangledown eff = unWrapT \circ hdlT \circ eff$$

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 $Free\ (\textit{Sig}_T + sig)$. The fact above follows from the universal property of free monads. It states that

[3] This restriction is reasonable because it covers most of the usual cases [35], except for *Maybe*.

for all signatures (functors) *Sign*, monads *M*, and polymorphic functions (natural transformations) $f :: Sign\ a \to M\ a$, 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 Sign* type.

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

Finally, given the transformer's run function $runT :: Monad \Rightarrow T\ m\ A \to m\ B$, we can derive an appropriate run function for the handler:

```
run :: Monad m ⇒ WrapT A m → m B
run = runT ∘ unWrapT
```

5. Case Study: Call/CC

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

5.1 Established Implementation

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

```
class Monad m ⇒ MonadCont m where
  callCC :: ((a → m b) → m a) → m a
```

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

```
instance MonadCont (Cont r) where
  callCC f = Cont (λk → runCont (f (λx → Cont (\_ → k x))) k)
```

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

```
newtype ContT r m a = CT {runCT :: (a → m r) → m r}
instance Monad m ⇒ MonadCont (ContT r m) where
  callCC f = CT (λk → runCT (f (λx → CT (\_ → k x))) k)
```

At first sight, 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 algebraic effects & handlers approach, and indeed, we are not aware of any existing implementation that offers *callCC*. Hence, it seems that monad transformers are more expressive on this account.

5.2 Reformulation

Nevertheless, following Thielecke [40] and Fiore & Staton [11], we can decompose *callCC* into two algebraic operations, given by the following *MonadJump* type class:

```
class Monad m ⇒ MonadJump ref m | m → ref where
  jump :: ref a → a → m b
  sub  :: (ref a → m b) → (a → m b) → 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 informal characterisation is captured in the following four laws,

$$\begin{aligned}
sub\ (\lambda r \to jump\ r\ x)\ k \quad &\equiv \quad k\ x \\
sub\ (\lambda\_ \to p)\ k \quad &\equiv \quad p \\
sub\ p\ (jump\ r') \quad &\equiv \quad p\ r' \\
sub\ (\lambda r_1 \to sub\ (\lambda r_2 \to p\ r_1\ r_2)\ (k_2\ r_1))\ k_1 \quad &\equiv \\
sub\ (\lambda r_2 \to sub\ (\lambda r_1 \to p\ r_1\ r_2)\ k_1)\ (sub\ k_2\ k_1) &
\end{aligned}$$

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

$$\begin{aligned}
jump\ r\ x \gg\!= k \quad &\equiv \quad jump\ r\ x \\
sub\ p\ q \gg\!= k \quad &\equiv \quad sub\ (p \ggg k)\ (q \ggg k)
\end{aligned}$$

The former expresses that a *jump* abandons the current continuation *k*. The latter expresses that both the main computation and the alternative computation constitute of the same common continuation *k* prefixed by respectively *p* and *q*.

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

```
callCC f = sub (λref → f (jump ref)) return
```

Here the *exit* mechanism is made explicit by the *jump* operation, which jumps to $return \ggg k \equiv k$, where *k* is the current continuation.

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

```
newtype Ref m a = ∀r. R {unRef :: a → m r}
jump (R exit) x = exit x ≫ return ⊥
sub k₁ k₂       = callCC (λexit → k₁ (R (k₂ >=> 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* ⊥ 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. We can easily establish one direction of this property with straightforward equational reasoning if we assume the following usual *callCC* property [6]:

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

which states that *exit* never returns.

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

5.3 Alternative Handler for *callCC*

The algebraic operations *jump* and *sub* give rise to the following signature:

```
data SUBST ref k = ∀a. JMP (ref a) a
                 | ∀a. SUB (ref a → k) (a → k)
```

for which we can easily derive a modular handler from the *ContT* monad transformer definition following the recipe of Section 4. 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.

```
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.

```
newtype b :← a = Switch {(₴) :: a → b}
```

The constructor $Switch :: (a \to b) \to (b :\leftarrow a)$ switches the direction of a function, and the deconstructor $(\reflectbox{\$}) :: (b :\leftarrow a) \to a \to b$ 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.

$$
\begin{aligned}
&\mathit{gen}_{SB} :: \mathit{Monad}\ m \Rightarrow r \rightarrow \mathit{Id}_H\ r\ m\\
&\mathit{gen}_{SB}\ x = \mathit{Id}_H\ (\mathit{return}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{alg}_{SB} :: \mathit{Monad}\ m \Rightarrow \textsc{Subst}\ ((:\leftarrow)\ (\mathit{Id}_H\ r\ m))\ (\mathit{Id}_H\ r\ m) \rightarrow \mathit{Id}_H\ r\ m\\
&\mathit{alg}_{SB}\ (\textsc{Jmp}\ \mathit{ref}\ x)\ = \mathit{ref} \mathbin{\$} x\\
&\mathit{alg}_{SB}\ (\textsc{Sub}\ k_1\ k_2) = k_1\ (\mathit{Switch}\ k_2)
\end{aligned}
$$

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 $\textsc{Subst}\ ((:\leftarrow)\ (\mathit{Id}_H\ r\ m))$. Hence, when we use the free monad transformer recipe of Section 4, this non-orthogonality carries over to the *run* function:

$$
\begin{aligned}
&\mathit{runSubstT} :: \mathit{Monad}\ m \Rightarrow \mathit{Free}_T\ (\textsc{Subst}\ ((:\leftarrow)\ (\mathit{Id}_H\ r\ m)))\ m\ r\\
&\qquad\qquad\qquad\qquad \rightarrow \mathit{Id}_H\ r\ m\\
&\mathit{runSubstT} = \mathit{foldFree}_T\ \mathit{gen}_{SB}\ \mathit{alg}_{SB}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{runSubstT}' :: \mathit{Monad}\ m \Rightarrow \mathit{Free}_T\ (\textsc{Subst}\ ((:\leftarrow)\ (\mathit{Id}_H\ r\ m)))\ m\ a\\
&\qquad\qquad\qquad\qquad \rightarrow ((a \rightarrow m\ r) \rightarrow \mathit{Id}_H\ r\ m)\\
&\mathit{runSubstT}'\ p\ k = \mathit{runSubstT}\ (p \ggg\!= \mathit{lift} \circ k)
\end{aligned}
$$

This brings us essentially back to the continuation monad transformer as $((a \rightarrow m\ r) \rightarrow \mathit{Id}_H\ m\ r) \cong \mathit{CodT}\ r\ m\ a$.

6. Related Work

There is a lot of related work that studies either monad transformers or algebraic effects and handlers separately.

6.1 Algebraic Effects and Handlers

Plotkin and Power were the first to explore effect operations [32], and gave an algebraic account of effects [33] and their combination [15]. Subsequently, Plotkin and Pretnar [34] have added the concept of handlers to deal with exceptions. This theoretical development has led to many language and library implementations.

Eff Perhaps the most prominent language is Eff [2], an OCaml-like language with native support for algebraic effects and handlers. It 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) $\mathit{Free}\ F\ A$; carriers of the form $S \rightarrow \mathit{Free}\ F\ B$ are encoded as $\mathit{Free}\ F\ (S \rightarrow \mathit{Free}\ F\ B)$. A special case are handlers that introduce new effects in the program, i.e., of type $\mathit{Free}\ (F + G)\ A \rightarrow \mathit{Free}\ (H + G)\ B$. We can model their modular carrier as $\mathit{Free}_T\ H$ since $\mathit{Free}_T\ H\ (\mathit{Free}\ G) \cong \mathit{Free}\ (H + G)$.

While Eff implicitly forwards operations that are not explicitly handled, its subtyping-based type system [3] 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 of handlers that rely on the presence of another effect, which can be expressed as relaxing $(\mathit{Monad}\ m) \Rightarrow m$ to $(\mathit{MonadEff}\ F\ m) \Rightarrow m$.

Handlers in Action Handlers in Action [22] builds on a formalisation similar to Eff's with carrier types essentially of the form $\mathit{Free}\ F\ A$ and a simple type system that provides neither subtyping nor parametric polymorphism. Yet, 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 $\mathit{Free}\ F\ A$, this implementation also supports carrier types of the form $S \rightarrow \mathit{Free}\ F\ A$. This work also introduces *shallow* handlers that only handle the first operation. We can model these as folds with tupling [14].

The implementation employs a *final* encoding [5] 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 (see [43] for an explanation of the latter).

Extensible Effects The Extensible Effects Haskell library [24] 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 [23] using an inlined co-Yoneda construction and a queue datatype [41]. The library provides modular *fold* recursion schemes for the $\mathit{Free}\ F\ A$ and $S \rightarrow \mathit{Free}\ F\ A$. 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 [4] 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 $\forall a. S_i \rightarrow M\ a$ and their composition yields a carrier $\forall a. (S_1, ..., S_n) \rightarrow M\ a$. Moreover, the handlers must share a common generator $(S_1, ..., S_n) \rightarrow A \rightarrow M\ B$.

Multicore OCaml comes equipped with algebraic effects and handlers intended to implement thread schedulers [8]. These handlers are similar to Eff's, but lack a type system.

The Frank language [27] shows many similarities with Eff, but allows arbitrary recursion patterns for handlers and even matching on multiple computations at the same time.

6.2 Monad Transformers

Moggi [28] used monads to model side-effects while working on computational models. Independently, Spivey [37] used monads while working on a theory of exceptions. Wadler [42] popularized monads in the context of Haskell, and others (e.g., [21, 38]) have sought to modularize them.

Monad transformers emerged [7, 25] from this process, and in later years various alternative implementation designs, facilitating monad (transformer) implementations, have been proposed, such as Filinksi's layered monads [10] 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 therefore can distinguish which operations are algebraic and which are not.

The administrative transformations on signature coproducts mentioned in Section 2.5 have also been studied for monad transformers by Schrijvers and Oliveira [36].

A very general way to combine monads is their coproduct [26], of which the coroduct of free monads used in the algebraic effects approach is a special case. Typically, the coproduct of two monads is too general to be direcly useful. Instead, the coproduct is post-processed with a mediating monad morphism into a monad that combines the two effects more usefully.

7. Conclusion

In this paper we have studied the use of transformers and algebraic effects to model a variety of modular effects. We have seen that monad transformers use monad subclass constraints to allow modular syntax, and that the semantics is given by a monad homomorphism. Algebraic effects provide modular signature functors whose semantics are given by folds over syntax trees.

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. The key is to identify modular carriers for handlers, and to notice that every transformer is such as modular carrier.

Acknowledgments

We are indebted to Ohad Kammar, Sam Lindley, Nicolas Oury, Oleg Kiselyov, Amr Sabry, Cameron Swords and Atze van der Ploeg for their excellent Haskell libraries of algebraic effects & handlers, which have been a major inspiration. We are grateful to Koen Claessen for pointing out relevant related work.

References

[1] R. Backhouse, M. Bijsterveld, R. Geldrop, and J. Woude. *Category Theory and Computer Science*, chapter Categorical fixed point calculus, pages 159–179. Springer, Berlin, Heidelberg, 1995.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers, 2012.

[3] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

[4] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, pages 133–144. ACM, 2013.

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

[6] M. Carlsson. Value recursion in the continuation monad. Unpublished note: http://www.carlssonia.org/ogi/mdo-callcc.pdf, Jan. 2003.

[7] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *CCTCS '93: Proceedings of the Conference on Category Theory and Computer Science*, 1993.

[8] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency with algebraic effects. 2015.

[9] S. Dolanand, L. Whiteand, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency with algebraic effects. In *OCaml Worshop*, 2015.

[10] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 175–188, 1999.

[11] M. P. Fiore and S. Staton. Substitution, jumps, and algebraic effects. In *Annual IEEE Symposium on Logic in Computer Science (LICS 2014)*, pages 41:1–41:10. IEEE Computer Society Press, July 2014.

[12] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. *SIGPLAN Not.*, 46(9):2–14, Sept. 2011.

[13] R. Hinze. Kan extensions for program optimisation or: Art and Dan explain an old trick. In *Mathematics of Program Construction*, volume 7342 of *LNCS*, pages 324–362. Springer, 2012.

[14] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[15] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[16] M. Jaskelioff. Modular monad transformers. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502, pages 64–79. Springer, 2009.

[17] M. Jaskelioff. Monatron: An extensible monad transformer library. In *Implementation and Application of Functional Languages*, volume 5836 of *LNCS*, pages 233–248. Springer, 2011.

[18] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441 – 4466, 2010.

[19] M. Jaskelioff and R. O'Connor. A representation theorem for second-order functionals. *J. Funct. Program.*, 25, 2015.

[20] M. Jaskelioff and E. Rivas. Functional pearl: a smart view on datatypes. In *ACM SIGPLAN International Conference on Functional Programming*, pages 355–361. ACM, 2015.

[21] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA, December 1993.

[22] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '14, pages 145–158. ACM, 2013.

[23] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 94–105. ACM, 2015.

[24] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013.

[25] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343. ACM, 1995.

[26] C. Lüth and N. Ghani. Composing monads using coproducts. In *ICFP '02*, pages 133–144. ACM, 2002.

[27] C. McBride. The Frank manual, May 2012.

[28] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[29] M. Piróg, N. Wu, and J. Gibbons. Modules over monads, and their algebras. In *International Conference on Algebra and Coalgebra in Computer Science*, Leibniz International Proceedings in Informatics, pages 287–300, 2015.

[30] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[31] G. Plotkin and M. Pretnar. *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Proceedings*, chapter Handlers of Algebraic Effects, pages 80–94. Springer, 2009.

[32] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.

[33] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[34] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[35] E. Rivas, M. Jaskelioff, and T. Schrijvers. From monoids to near-semirings: the essence of MonadPlus and Alternative. In *International Symposium on Principles and Practice of Declarative Programming*, pages 196–207. ACM, 2015.

[36] T. Schrijvers and B. C. d. S. Oliveira. Monads, zippers and views : virtualizing the monad stack. *ACM SIGPLAN NOTICES*, 2011.

[37] M. Spivey. A functional theory of exceptions. *Sci. Comput. Program.*, 14(1):25–42, May 1990.

[38] G. L. Steele, Jr. Building interpreters by composing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 472–492, 1994.

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

[40] H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

[41] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *ACM SIGPLAN symposium on Haskell*, pages 133–144, 2014.

[42] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[43] N. Wu and T. Schrijvers. Fusion for free - efficient algebraic effect handlers. In *Mathematics of Program Construction*, volume 9129 of *LNCS*, pages 302–322. Springer, 2015.

[44] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, 2014. ACM.

A. Equivalence of Algebraic operations

For any signature *sig* and monad *m*, operations *eff* :: ∀*a*. *sig a* → *m a* are in one-to-one correspondence with operations *op* :: *sig* (*m a*) → *m a*, such that the *join* of the monad is respected

$$join \circ op = op \circ fmap\ join \tag{8}$$

The correspondence of algebraic operations is witnessed by the following functions:

```
algEff :: MonadEff sig m ⇒ sig (m a) → m a
algEff = join ∘ eff
fromAlg :: (Functor sig, Monad m)
    ⇒ (∀a. sig (m a) → m a) → (sig a → m a)
fromAlg op = op ∘ fmap return
```

One direction of the proof is the following:

```
   fromAlg algEff
=    { definition of fromAlg and algEff }
   join ∘ eff ∘ fmap return
=    { naturality of eff }
   join ∘ fmap return ∘ eff
=    { monad law (2) }
   eff
```

In the other direction, let us assume that *eff* = *fromAlg op*, and that *op* respects the *join* of the monad (i.e. has property (8)).

```
   algEff
=    { definition of algEff }
   join ∘ eff
=    { assumption }
   join ∘ fromAlg op
=    { definition of fromAlg }
   join ∘ op ∘ fmap return
=    { property (8) }
   op ∘ fmap join ∘ fmap return
=    { functors, monad law (1) }
   op
```

B. More on Handlers as Transformers

Now, we give a categorical explanation and a generalisation of the constructions given in Section 3. To distinguish between Haskell code and the more general constructions, we use a categorical notation. We denote functors as $F, M, L, \ldots$, and their composition by juxtaposition, for example ML. We use the letter Σ to denote endofunctors that represent signatures.

We abuse the notation by identifying monads with their underlying endofunctor. We use $\eta : \mathsf{Id} \to M$ for the unit (*return*), and $\mu : MM \to M$ for the multiplication (*join*). If there is more than one monad in the context, we use superscripts to assign the natural transformations to appropriate monads, for example $\eta^M : \mathsf{Id} \to M$ and $\mu^M : MM \to M$.

We denote the (algebraically) free monad generated by an endofunctor Σ as Σ^*. Its unit and multiplication are denoted as η^{F} and μ^{F} respectively. The transformation that adds a new layer to the monad (the action of the free Σ-algebra) is called $\mathsf{cons} : \Sigma\Sigma^* \to \Sigma^*$.

B.1 Background: The Free Monad Transformer

The free monad transformer (aka the resumption monad) was introduced by Moggi [28]. The properties shown in this subsection were discussed by Hyland, Plotkin, and Power [15].

The carrier of the free monad transformer of an endofunctor Σ and a monad M is given using initial algebras as $A \mapsto \mu X. M(\Sigma X + A)$. Alternatively, using the *rolling lemma* [1], it can be given as the composition $M(\Sigma M)^*$.

The category of Eilenberg–Moore algebras for $M(\Sigma M)^*$ can be described as follows:

- Objects are tuples $(A,\ f : \Sigma A \to A,\ m : MA \to A)$, where m is an Eilenberg–Moore algebra.
- A morphism between (A, f, m) and (B,g,n) is given by a morphism $h : A \to B$ such that $h \cdot f = g \cdot \Sigma h$ and $h \cdot m = n \cdot Mh$.

The free object in this category generated by an object A is given as $(M(\Sigma M)^*A,\ \mathsf{algF},\ \mathsf{algM})$, where:

$$\mathsf{algF} = (\Sigma M(\Sigma M)^*A \xrightarrow{\mathsf{cons}} (\Sigma M)^*A \xrightarrow{\eta^M} M(\Sigma M)^*A)$$

$$\mathsf{algM} = (MM(\Sigma M)^*A \xrightarrow{\mu^M} M(\Sigma M)^*A)$$

The freeness property in this case means that given any other algebra (A, f, m), there exists a unique Eilenberg–Moore algebra $\langle\!\langle f, m \rangle\!\rangle : M(\Sigma M)^*A \to A$ that makes the following diagram commute:

$$\begin{tikzcd}
\Sigma A \arrow[r, "\Sigma\eta_A"] \arrow[rrd, "f"'] & \Sigma M(\Sigma M)^*A \arrow[r, "\mathsf{algF}"] & M(\Sigma M)^*A \arrow[d, dashed, "\langle\!\langle f,m \rangle\!\rangle"] & MM(\Sigma M)^*A \arrow[l, "\mathsf{algM}"'] & MA \arrow[l, "M\eta_A"'] \arrow[lld, "m"] \\
 & & A & &
\end{tikzcd}$$

Let F be an endofunctor. Consider two natural transformations $\phi : \Sigma F \to F$ and $\psi : MF \to F$ such that each component of ψ is an Eilenberg–Moore algebra (in other words, F is a left module over M with the action given as ψ). Then, the family of morphisms $\langle\!\langle \phi, \psi \rangle\!\rangle_A = \langle\!\langle \phi_A, \psi_A \rangle\!\rangle$ is a natural transformation $M(\Sigma M)^*F \to F$.

B.2 Background: The Codensity Monad

Now, we generalise the results from Section 3. Instead of working with modular carriers, we work with left modules [29]. A left module over a monad M is an endofunctor S together with a natural transformation $\delta : MS \to M$ such that $\delta \cdot M\delta : MMS \to S$ and $\delta \cdot \eta S : S \to S$.

We generalise the continuation monad to the codensity monad:

```
newtype Cod f a = Cod (∀x. (a → f x) → f x)
runCod k (Cod t) = t k
instance Monad (Cod f) where
  return x    = Cod (λk → k x)
  (Cod p) ≫= f = Cod (λk → p (runCod k ∘ f))
```

First, we develop some theory of such codensity monads generated by modules over monads. Then, we discuss how this setting instantiates to the continuation monad with the answer type given by a modular carrier.

Instead of reasoning directly about Haskell code, we take a more abstract approach employing right Kan extensions. We benefit from Hinze's [13] string diagrams for Kan extensions.

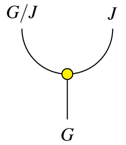
Let $\mathscr{C}$, $\mathscr{D}$, and $\mathscr{E}$ be categories. A *right Kan extension* of a functor $G : \mathscr{C} \to \mathscr{E}$ along a functor $J : \mathscr{C} \to \mathscr{D}$ consists of:

- a functor $G/J : \mathscr{D} \to \mathscr{E}$,
- a natural transformation $\mathsf{run} : (G/J)J \to G$

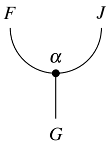
such that for all functors $F : \mathscr{D} \to \mathscr{E}$ and natural transformations $\alpha : FJ \to G$, there exists a unique natural transformation $[\alpha] : F \to G/J$ (called the *shift* of α) with the property

$$\mathsf{run} \cdot [\alpha]J = \alpha \tag{10}$$

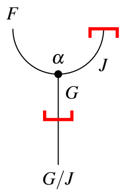
In the string diagram notation, we denote run using a yellow dot:



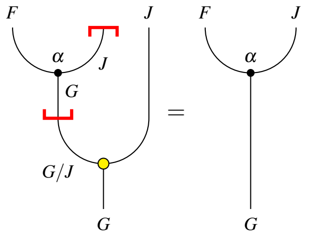
Given a natural transformation $\alpha : FJ \to G$



we denote $[\alpha] : F \to G/J$ using brackets:

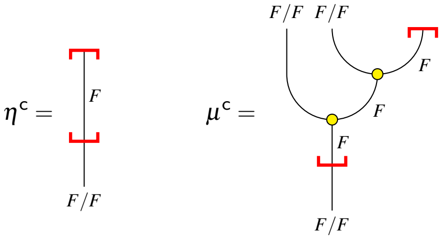


The property (10) can be expressed using string diagrams as:



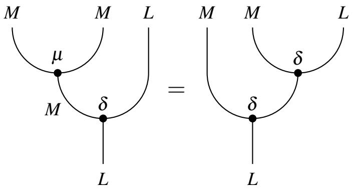
(11)

The *codensity monad* of a functor $F : \mathscr{D} \to \mathscr{C}$ is given by the right Kan extension F/F. The unit is given by $\eta^c = [\mathsf{id}]$, while the multiplication is given by $\mu^c = [\mathsf{run} \cdot (F/F)\mathsf{run}]$. Using string diagrams, they can be expressed as:

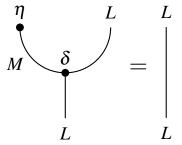


B.3 The Codensity Monad of a Left Module

We fix a monad M. Let L be a left module over M. This means that L is an endofunctor and there exists a natural transformation $\delta : ML \to L$ such that the following equalities hold:



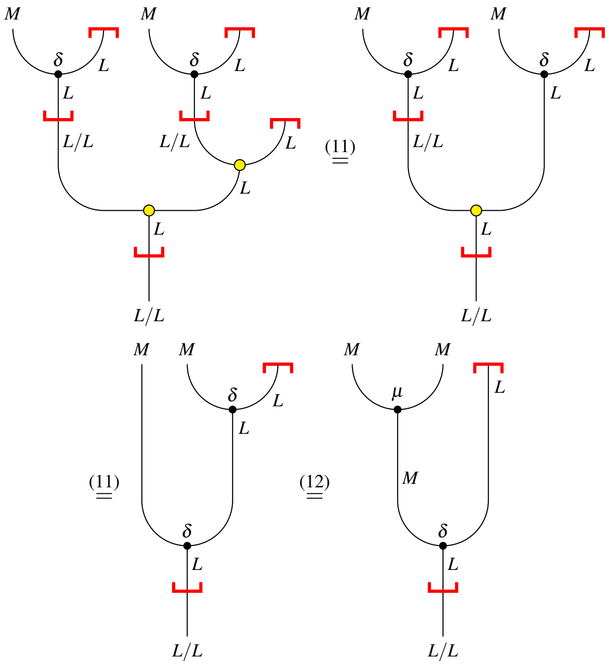
(12)



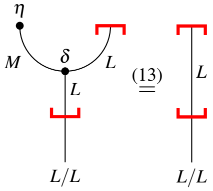
(13)

Theorem 1. *The natural transformation $[\delta] : M \to L/L$ is a monad morphism.*

Proof. Preservation of the multiplication:



Preservation of the unit:



□

Corollary 2. *If L is a left module over M, so is L/L.*

Now, the results given in Section 3 can be recovered simply by noticing that, for a modular carrier h and a monad m, the type $h\,m$ is a module over m understood as a constant endofunctor. The discussed monad morphism is the *lift* operation for the *ContH* transformer.

B.4 The Free Monad Transformer vs The Codensity Monad

We generalise the type class of modular carriers to parametrised left modules:

```
class PLModule m l where
  joinl' :: Monad m ⇒ m (l a) → l a
```

We also introduce a wrapper for the codensity monad, so that we can parametrise it with a monad:

newtype $\mathit{HCod}\ h\ (m :: * \to *)\ a = \mathit{HCod}\ (\forall x.\,(a \to h\ m\ x) \to h\ m\ x)$

Note that, according to Theorem 1, for all parametrised left modules h, the type $HCod\ h$ is a monad transformer.

Now, we generalise the results connecting handlers with the codensity monad from Section 3 to the free monad transformer instead of the free monad of a coproduct of signatures. In other words, we define the following morphism:

$$\begin{aligned}
&freeTToHCod :: (Functor\ sig_1, Monad\ m, PLModule\ h)\\
&\qquad \Rightarrow (\forall x.\, sig_1\ (h\ m\ x) \to h\ m\ x)\\
&\qquad \to Free_T\ sig_1\ m\ a \to HCod\ h\ m\ a\\
&freeTToHCod\ alg\ f = \ldots
\end{aligned}$$

We show that it is a monad morphism and that the following diagram commutes for any algebra *alg* and a generator *gen*:

$$\begin{tikzcd}
Free_T\ sig_1\ m\ a \arrow[rr, "freeTToHCod\ alg"] \arrow[dr, "fold\ gen\ (liftAlg\ alg)"'] & & HCod\ h\ m\ a \arrow[dl, "flip\ (runCod \circ runHCod)\ gen"] \\
& h\ m\ a &
\end{tikzcd}$$

From the categorical perspective, we fix a monad M, a left module L with the action $\delta : ML \to L$, and an endofunctor Σ.

Theorem 3. *Given a natural transformation $\alpha : \Sigma L \to L$, the endofunctor L is a left module over $M(\Sigma M)^*$ with the action defined as $\langle\!\langle \alpha, \delta \rangle\!\rangle : M(\Sigma M)^* L \to L$.*

Proof. The result follows from the fact that $\langle\!\langle$-,-$\rangle\!\rangle$ is an Eilenberg–Moore algebra for the monad $M(\Sigma M)^*$. □

We model *freeTToHCod* as in the following:

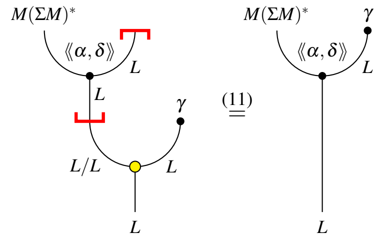
Corollary 4. *Given a natural transformation $\alpha : \Sigma L \to L$, the morphism $[\langle\!\langle \alpha, \delta \rangle\!\rangle] : M(\Sigma M)^* \to L/L$ is a monad morphism.*

Proof. Apply Theorems 1 and 3. □

Theorem 5. *Given a natural transformation $\gamma : \mathsf{Id} \to L$ (a generator), the following diagram commutes:*

$$\begin{tikzcd}
M(\Sigma M)^* \arrow[r, "{[\langle\!\langle \alpha, \delta \rangle\!\rangle]}"] \arrow[d, "M(\Sigma M)^*\gamma"'] & L/L \arrow[r, "(L/L)\gamma"] & (L/L)L \arrow[d, "\mathsf{run}"] \\
M(\Sigma M)^* L \arrow[rr, "\langle\!\langle \alpha, \delta \rangle\!\rangle"] & & L
\end{tikzcd}$$

Proof. Using string diagrams:


$M(\Sigma M)^*$
$\langle\!\langle \alpha, \delta \rangle\!\rangle$
L
L
γ
L/L
L
L
$\stackrel{(11)}{=}$
$M(\Sigma M)^*$
γ
$\langle\!\langle \alpha, \delta \rangle\!\rangle$
L
L

□

C. Monad Transformers as Modular Carriers

We show that $WrapT\ a$ is a modular carrier for any type a, that is, we show that the modular carrier laws hold. The first law (6) shows coherence with *return* (for readability, we omit the *WrapT* and *unWrapT* isomorphisms):

$$\begin{aligned}
&\ joinl \circ return\\
=&\quad \{\ \text{definition } joinl\ \}\\
&\ join \circ lift \circ return\\
=&\quad \{\ lift \text{ is a monad homomorphism (4)}\ \}\\
&\ join \circ return\\
=&\quad \{\ \text{monad law (1)}\ \}\\
&\ id
\end{aligned}$$

The second law (7) is similar, and shows coherence with *join*:

$$\begin{aligned}
&\ joinl \circ join\\
=&\quad \{\ \text{definition of } joinl\ \}\\
&\ join \circ lift \circ join\\
=&\quad \{\ lift \text{ is a monad homomorphism (5)}\ \}\\
&\ join \circ join \circ lift \circ fmap\ lift\\
=&\quad \{\ \text{monad law (3)}\ \}\\
&\ join \circ fmap\ join \circ lift \circ fmap\ lift\\
=&\quad \{\ lift \text{ is natural}\ \}\\
&\ join \circ lift \circ fmap\ (join \circ lift)\\
=&\quad \{\ \text{definition of } joinl\ \}\\
&\ joinl \circ fmap\ joinl
\end{aligned}$$