Fusion for Free

Efficient Algebraic Effect Handlers

Nicolas Wu[1] and Tom Schrijvers[2]

[1] Department of Computer Science, University of Bristol
[2] Department of Computer Science, KU Leuven

Abstract. Algebraic effect handlers are a recently popular approach for modelling side-effects that separates the syntax and semantics of effectful operations. The shape of syntax is captured by functors, and free monads over these functors denote syntax trees. The semantics is captured by algebras, and effect handlers pass these over the syntax trees to interpret them into a semantic domain.

This approach is inherently modular: different functors can be composed to make trees with richer structure. Such trees are interpreted by applying several handlers in sequence, each removing the syntactic constructs it recognizes. Unfortunately, the construction and traversal of intermediate trees is painfully inefficient and has hindered the adoption of the handler approach.

This paper explains how a sequence of handlers can be fused into one, so that multiple tree traversals can be reduced to a single one and no intermediate trees need to be allocated. At the heart of this optimization is keeping the notion of a free monad abstract, thus enabling a change of representation that opens up the possibility of fusion. We demonstrate how the ensuing code can be inlined at compile time to produce efficient handlers.

1 Introduction

Free monads are currently receiving a lot of attention. They are at the heart of *algebraic effect handlers*, a new purely functional approach for modelling side effects introduced by Plotkin and Power [15]. Much of their appeal stems from the separation of the syntax and semantics of effectful operations. This is both conceptually simple and flexible, as multiple different semantics can be provided for the same syntax.

The syntax of the primitive side-effect operations is captured in a signature functor. The free monad over this functor assembles the syntax for the individual operations into an abstract syntax tree for an effectful program. The semantics of the individual operations is captured in an algebra, and an effect handler folds the algebra over the syntax tree of the program to interpret it into a semantic domain.

A particular strength of the approach is its inherent modularity. Different signature functors can be composed to make trees with richer structure. Such trees are interpreted by applying several handlers in sequence, each removing the syntactic constructs it recognizes.

Unfortunately, the construction and traversal of intermediate trees is rather costly. This inefficiency is perceived as a serious weakness of effect handlers, especially when compared to the traditional approach of composing effects with monad transformers. While several authors address the cost of constructing syntax trees with free monads, efficiently applying multiple handlers in sequence has received very little attention. As far as we know, only Kammar et al. [9] provide an efficient implementation. Unfortunately, this implementation does not come with an explanation. Hence it is underappreciated and ill-understood.

In this paper we close the gap and explain how a sequence of algebraic effect handlers can be effectively fused into a single handler. Central to the paper are the many interpretations of the word *free*. Interpreting *free* monads as the initial objects of the more general term algebras and, in particular, term monads provide an essential change of perspective where *free* theorems enable fusion. The codensity monad facilitates the way, turning any term algebra into a monadic one for *free*, and with an appropriate code setup in Haskell the GHC compiler takes care of fusion at virtually no cost[3] to the programmer. The result is an effective implementation that compares well to monad transformers.

2 Algebraic Effect Handlers

The idea of the algebraic effect handlers approach is to consider the free monad over a particular functor as an abstract syntax tree (AST) for an effectful computation. The functor is used to generate the nodes of a free structure whose leaves correspond to variables. This can be defined as an inductive datatype *Free f* for a given functor *f*.

```
data Free f a where
  Var :: a → Free f a
  Con :: f (Free f a) → Free f a
```

The nodes are *con*structed by *Con*, and the *var*iables are given by *Var*.

Since a value of type *Free f a* is an inductive structure, we can define a *fold* for it by providing a function *gen* that deals with generation of values from *Var x*, and an algebra *alg* that is used to recursively collapse an operation *Con op*.

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

Algebraic effect handlers give a semantics to the syntax tree: one way of doing this is by using a *fold*.

[3] Yes, almost for *free*!

The behaviour of folds when composed with other functions is described by fusion laws. The first law describes how certain functions that are precomposed with a fold can be incorporated into a new fold:

$$\mathit{fold}\ \mathit{alg}\ \mathit{gen} \cdot \mathit{fmap}\ h = \mathit{fold}\ \mathit{alg}\ (\mathit{gen} \cdot h) \tag{1}$$

The second law shows how certain functions that are postcomposed with a fold can be incorporated into a new fold:

$$k \cdot \mathit{fold}\ \mathit{alg}\ \mathit{gen} = \mathit{fold}\ \mathit{alg}'\ (k \cdot \mathit{gen}) \tag{2}$$

this is subject to the condition that $k \cdot \mathit{alg} = \mathit{alg}' \cdot \mathit{fmap}\ k$.

The monadic instance of the free monad is witnessed by the following:

```
instance Functor f ⇒ Monad (Free f) where
  return x = Var x
  m >>= f  = fold Con f m
```

Variables are the way of providing a return for the monad, and extending a syntax tree by means of a function f corresponds to applying that function to the variables found at the leaves of the tree.

2.1 Nondeterminsm

A functor supplies the abstract syntax for the primitive effectful operations in the free monad. For instance, the *Nondet* functor provides the *Or k k* syntax for a binary nondeterministic choice primitive. The parameter to the constructor of type k marks the recursive site of syntax, which indicates where the continuation is after this syntactic fragment has been evaluated.

```
data Nondet k where
  Or :: k → k → Nondet k
instance Functor Nondet where
  fmap f (Or x y) = Or (f x) (f y)
```

This allows us to express the syntax tree of a computation that nondeterministically returns *True* or *False*.

```
coin :: Free Nondet Bool
coin = Con (Or (Var True) (Var False))
```

The syntax is complemented by semantics in the form of effect handlers—functions that replace the syntax by values from a semantic domain. Using a *fold* for the free monad is a natural way of expressing such functions. For instance, here is a handler that interprets *Nondet* in terms of lists of possible outcomes:

$$\mathit{handle}_{\mathit{Nondet}_{[]}} :: \mathit{Free}\ \mathit{Nondet}\ a \to [a]$$
$$\mathit{handle}_{\mathit{Nondet}_{[]}} = \mathit{fold}\ \mathit{alg}_{\mathit{Nondet}_{[]}}\ \mathit{gen}_{\mathit{Nondet}_{[]}}$$

where $alg_{Nondet_{[]}}$ is the *Nondet*-algebra that interprets terms constructed by *Or* operations and $gen_{Nondet_{[]}}$ interprets variables.

$$alg_{Nondet_{[]}} :: Nondet\ [a] \to [a]$$
$$alg_{Nondet_{[]}}\ (Or\ l_1\ l_2) = l_1 +\!\!+\ l_2$$
$$gen_{Nondet_{[]}} :: a \to [a]$$
$$gen_{Nondet_{[]}}\ x = [x]$$

The variables of the syntax tree are turned into singleton lists by $gen_{Nondet_{[]}}$, and choices between alternatives are put together by $alg_{Nondet_{[]}}$, which appends lists.

As an example, we can interpret the *coin* program:

```
> handle_Nondet[] coin
[True, False]
```

This particular interpretation gives us a list of the possible outcomes.

Generalizing away from the details, handlers are usually presented in the form

$$hdl :: \forall a\ .\ Free\ F\ a \to H\ a$$

where F and H are arbitrary functors determined by the handler.

2.2 Handler Composition

There are many useful scenarios that involve the (function) composition of effect handlers. We now consider two classes of this kind of composition.

Effect Composition A first important class of scenarios is where multiple effects are combined in the same program. To this end we compose signatures and handlers.

The coproduct functor $f + g$ makes it easy to compose functors:

```
data (+) f g a where
  Inl :: f a → (f + g) a
  Inr :: g a → (f + g) a
instance (Functor f, Functor g) ⇒ Functor (f + g) where
  fmap f (Inl s) = Inl (fmap f s)
  fmap f (Inr s) = Inr (fmap f s)
```

The free monad of a coproduct functor is a tree where each node can be built from syntax from either f or g.

Composing handlers is easy too: if the handlers are written in a compositional style, then function composition does the trick. A compositional handler for the functor F has a signature of the form:

$$hdl :: \forall g\ a\ .\ Free\ (F + g)\ a \to H_1\ (Free\ g\ (G_1\ a))$$

This processes only the F-nodes in the AST and leaves the g-nodes as they are. Hence the result of the compositional handler is a new (typically smaller) AST with only g-nodes. The variables of type $G_1\ a$ in the resulting AST are derived from the variables of type a in original AST as well as from the processed operations. Moreover, the new AST may be embedded in a context H_1.

For instance, the compositional nondeterminism handler is defined as follows, where $F = \mathit{Nondet}$, $G_1 = [\,]$ and implicitly $H_1 = \mathsf{Id}$.

$$\begin{array}{l} \mathit{handle}_{\mathit{Nondet}} :: \mathit{Functor}\ g \Rightarrow \mathit{Free}\ (\mathit{Nondet} + g)\ a \rightarrow \mathit{Free}\ g\ [a] \\ \mathit{handle}_{\mathit{Nondet}} = \mathit{fold}\ (\mathit{alg}_{\mathit{Nondet}} \triangledown \mathit{Con})\ \mathit{gen}_{\mathit{Nondet}} \end{array}$$

Here the variables are handled with the monadified version of $\mathit{gen}_{\mathit{Nondet}_{[]}}$, given by $\mathit{gen}_{\mathit{Nondet}}$.

$$\begin{array}{l} \mathit{gen}_{\mathit{Nondet}} :: \mathit{Functor}\ g \Rightarrow a \rightarrow \mathit{Free}\ g\ [a] \\ \mathit{gen}_{\mathit{Nondet}}\ x = \mathit{Var}\ [x] \end{array}$$

The g nodes are handled by a Con algebra, which essentially leaves them untouched. The Nondet nodes are handled by the $\mathit{alg}_{\mathit{Nondet}}$ algebra, which is a monadified version of $\mathit{alg}_{\mathit{Nondet}_{[]}}$.

$$\begin{array}{l} \mathit{alg}_{\mathit{Nondet}} :: \mathit{Functor}\ g \Rightarrow \mathit{Nondet}\ (\mathit{Free}\ g\ [a]) \rightarrow \mathit{Free}\ g\ [a] \\ \mathit{alg}_{\mathit{Nondet}}\ (\mathit{Or}\ \mathit{ml}_1\ \mathit{ml}_2) = \\ \quad \mathbf{do}\ \{ l_1 \leftarrow \mathit{ml}_1;\ l_2 \leftarrow \mathit{ml}_2;\ \mathit{Var}\ (l_1 +\!\!+ l_2) \} \end{array}$$

The junction combinator $(\triangledown)$ composes the algebras for the two kinds of nodes.

$$\begin{array}{l} (\triangledown) :: (f\ b \rightarrow b) \rightarrow (g\ b \rightarrow b) \rightarrow ((f + g)\ b \rightarrow b) \\ (\triangledown)\ \mathit{alg}_f\ \mathit{alg}_g\ (\mathit{Inl}\ s) = \mathit{alg}_f\ s \\ (\triangledown)\ \mathit{alg}_f\ \mathit{alg}_g\ (\mathit{Inr}\ s) = \mathit{alg}_g\ s \end{array}$$

In the definition of *handlerNondet*, we use $\mathit{alg}_{\mathit{Nondet}} \triangledown \mathit{Con}$. Since the functor in question is $\mathit{Nondet} + g$, the values constructed by Nondet are handled by $\mathit{alg}_{\mathit{Nondet}}$, and values constructed by g are left untouched: the fold unwraps one level of Con, but then replaces it with a Con again.

A second example of an effect signature is that of state, whose primitive operations *Get* and *Put* respectively query and modify the implicit state.

$$\begin{array}{l} \mathbf{data}\ \mathit{State}\ s\ k\ \mathbf{where} \\ \quad \mathit{Put} :: s \rightarrow k \rightarrow \mathit{State}\ s\ k \\ \quad \mathit{Get} :: (s \rightarrow k) \rightarrow \mathit{State}\ s\ k \\ \mathbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \mathbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Put}\ s\ k) = \mathit{Put}\ s\ (f\ k) \\ \quad \mathit{fmap}\ f\ (\mathit{Get}\ k) \quad = \mathit{Get}\ (f \cdot k) \end{array}$$

The compositional handler for state is as follows, where $F = \mathit{State}\ s$, $H_1 = s \rightarrow -$ and implicitly $G_1 = \mathsf{Id}$.

$$handle_{State} :: Functor\ g \Rightarrow Free\ (State\ s + g)\ a \rightarrow (s \rightarrow Free\ g\ a)$$
$$handle_{State} = fold\ (alg_{State} \triangledown con_{State})\ gen_{State}$$

This time the variable and constructor cases are defined as:

$$gen_{State} :: Functor\ g \Rightarrow a \rightarrow (s \rightarrow Free\ g\ a)$$
$$gen_{State}\ x\ s = Var\ x$$
$$alg_{State} :: Functor\ g \Rightarrow State\ s\ (s \rightarrow Free\ g\ a) \rightarrow (s \rightarrow Free\ g\ a)$$
$$alg_{State}\ (Put\ s'\ k)\ s = k\ s'$$
$$alg_{State}\ (Get\ k)\quad s = k\ s\ s$$

Using gen_{State}, a variable x is replaced by a version that ignores any incoming state parameter s. Any stateful constructs are handled by alg_{State}, where a continuation k proceeds by using the appropriate state: if the syntax is a $Put\ s'\ k$, then the new state is s', otherwise the syntax is $Get\ k$, in which case the state is left unchanged and passed as s.

Finally, any syntax provided by g is adapted by con_{State} to take the extra state parameter s into account:

$$con_{State} :: Functor\ g \Rightarrow g\ (s \rightarrow Free\ g\ a) \rightarrow (s \rightarrow Free\ g\ a)$$
$$con_{State}\ op\ s = Con\ (fmap\ (\lambda m \rightarrow m\ s)\ op)$$

This feeds the state s to the continuations of the operation.

To demonstrate effect composition we can put *Nondet* and *State* together and handle them both. Before we do so, we also need a base case for the composition, which is the empty signature *Void*.

data *Void k*

instance *Functor Void*

The *Void* handler only provides a variable case since the signature has no constructors. In fact, a *Free Void* term can only be a $Var\ x$, so x is immediately output using the identity function.

$$handle_{Void} :: Free\ Void\ a \rightarrow a$$
$$handle_{Void} = fold\ \bot\ id$$

Finally, we can put together a composite handler for programs that feature both nondeterminism and state. The signature of such programs is the composition of the three basic signatures:

type $\Sigma = Nondet + (State\ Int + Void)$

The handler is the composition of the three handlers, working from the left-most functor in the signature:

$$handle_{\Sigma} :: Free\ \Sigma\ a \rightarrow Int \rightarrow [a]$$
$$handle_{\Sigma}\ prog = handle_{Void} \cdot (handle_{State} \cdot handle_{Nondet})\ prog$$

Effect Delegation Another important class of applications are those where a handler expresses the complex semantics of particular operations in terms of more primitive effects.

For instance, the following logging handler for state records every update of the state by means of the *Writer* effect.

$$\begin{aligned}
&handle_{LogState} :: Free\ (State\ s)\ a \to s \to Free\ (Writer\ String + Void)\ a\\
&handle_{LogState} = fold\ alg_{LogState}\ gen_{State}\\
&alg_{LogState} :: State\ s\ (s \to Free\ (Writer\ String + Void)\ a)\\
&\qquad\qquad \to s \to Free\ (Writer\ String + Void)\ a\\
&alg_{LogState}\ (Put\ s'\ k)\ s = Con\ (Inl\ (Tell\ \texttt{"put"}\ (k\ s')))\\
&alg_{LogState}\ (Get\ k)\quad\ s = k\ s\ s
\end{aligned}$$

The syntax of the *Writer* effect is captured by the following functor, where w is a parameter that represents the type of values that are written to the log:

$$\begin{aligned}
&\mathbf{data}\ Writer\ w\ k\ \mathbf{where}\\
&\quad Tell :: w \to k \to Writer\ w\ k\\
&\mathbf{instance}\ Functor\ (Writer\ w)\ \mathbf{where}\\
&\quad fmap\ f\ (Tell\ w\ k) = Tell\ w\ (f\ k)
\end{aligned}$$

A semantics can be given by the following handler, where w is constrained to be a member of the *Monoid* typeclass.

$$\begin{aligned}
&handle_{Writer} :: (Functor\ g, Monoid\ w) \Rightarrow Free\ (Writer\ w + g)\ a \to Free\ g\ (w, a)\\
&handle_{Writer} = fold\ (alg_{Writer} \nabla Con)\ gen_{Writer}
\end{aligned}$$

The variables are evaluated by pairing with the unit of the monoid given by *mempty* before being embedded into the monad m_2.

$$\begin{aligned}
&gen_{Writer} :: (Monad\ m_2, Monoid\ w) \Rightarrow a \to m_2\ (w, a)\\
&gen_{Writer}\ x = return\ (mempty, x)
\end{aligned}$$

When a $Tell\ w_1\ k$ operation is encountered, the continuation k is followed by a state where w_1 is appended using *mappend* to any generated logs.

$$\begin{aligned}
&alg_{Writer} :: (Monad\ m_2, Monoid\ w) \Rightarrow Writer\ w\ (m_2\ (w, a)) \to m_2\ (w, a)\\
&alg_{Writer}\ (Tell\ w_1\ k) = k \ggg \lambda(w_2, x) \to return\ (w_1\ \text{`}mappend\text{`}\ w_2, x)
\end{aligned}$$

To see this machinery in action, consider the following program that makes use of state:

$$\begin{aligned}
&program :: Int \to Free\ (State\ Int)\ Int\\
&program\ n\\
&\quad \mid n \leqslant 0 \qquad\ \ = Con\ (Get\ var)\\
&\quad \mid otherwise = Con\ (Get\ (\lambda s \to Con\ (Put\ (s + n)\ (program\ (n - 1)))))
\end{aligned}$$

This is then simply evaluated by running handlers in sequence.

$$example :: Int \to (String, Int)$$
$$example\ n = (handle_{Void} \cdot handle_{Writer} \cdot handle_{LogState}\ (program\ n))\ 0$$

To fully interpret a stateful program, we must first run $handle_{LogState}$, which interprets the *Tell* operations by generating a tree with *Writer String* syntax. This generated syntax is then handled with the $handle_{Writer}$ handler.

3 Fusion

The previous composition examples lead us to the main challenge of this paper: The composition of two handlers produces an intermediate abstract syntax tree. How can we fuse the two handlers into a single one that does not involve an intermediate tree?

More concretely, given two handlers of the form:

$$handler_1 :: Free\ F_1\ a \to H_1\ (Free\ F_2\ (G_1\ a))$$
$$handler_1 = fold\ alg_1\ gen_1$$
$$handler_2 :: Free\ F_2\ a \to H_2\ a$$
$$handler_2 = fold\ alg_2\ gen_2$$

where F_1 and F_2 are signature functors and H_1, G_1 and H_2 are arbitrary functors, our goal is to obtain a combined handler

$$pipeline_{12} :: Free\ F_1\ a \to H_1\ (H_2\ (G_1\ a))$$
$$pipeline_{12} = fold\ alg_{12}\ gen_{12}$$

such that

$$fmap\ handler_2 \cdot handler_1 = pipeline_{12} \quad (3)$$

3.1 Towards Proper Builders

The fact that $handler_1$ *builds* an AST over functor F_2 and that $handler_2$ *folds* over this AST suggests a particular kind of fusion known as shortcut fusion or fold/build fusion [4].

One of the two key ingredients for this kind of fusion is already manifestly present: the *fold* in $handler_2$. Yet, the structure and type of $handler_1$ do not necessarily force it to be a proper builder: fold/build fusion requires that the builder creates the F_2-AST from scratch by generating all the *Var* and *Con* constructors itself. Indeed, in theory, $handler_1$ could produce the F_2-AST out of ready-made components supplied by (fields of) a colluding F_1 functor.

In order to force $handler_1$ to be a proper builder, we require it to be implemented against a builder interface rather than a concrete representation. This builder interface is captured in the typeclass *TermMonad* (explained below), and then, with the following constraint polymorphic signature, $handler_1$ is guaranteed to build properly:

$$handler_1 :: (TermMonad\ m_2\ F_2) \Rightarrow Free\ F_1\ a \to H_1\ (m_2\ (G_1\ a))$$

Term Algebras The concept of a *term algebra* provides an abstract interface for the primitive ways to build an AST: the two constructors *Con* and *Var* of the free monad. We borrow the nomenclature from the literature on universal algebras [2].

A *term algebra* is an f-algebra $con :: f\ (h\ a) \rightarrow h\ a$ with a carrier $h\ a$. The values in $h\ a$ are those generated by the set of variables a with a valuation function $var :: a \rightarrow h\ a$, as well as those that arise out of repeated applications of the algebra. This is modelled by the typeclass *TermAlgebra* $h\ f$ as follows:

```
class Functor f ⇒ TermAlgebra h f | h → f where
  var :: ∀a . a → h a
  con :: ∀a . f (h a) → h a
```

The function *var* is used to embed a *var*iable into the term, and the function *con* is used to *con*struct a term from existing ones. This typeclass is well-defined only when $h\ a$ is indeed generated by *var* and *con*.

The most trivial instance of this typeclass is of course that of the free monad.

```
instance Functor f ⇒ TermAlgebra (Free f) f where
  var = Var
  con = Con
```

Term Monads There are two additional convenient ways to build an AST: the monadic primitives *return* and $(\gg\!=)$.

A monad m is a *term monad* for a functor f, if it there is a term algebra for f whose carrier is $m\ a$. We can model this relationship as a typeclass with no members.

```
class (Monad m, TermAlgebra m f) ⇒ TermMonad m f | m → f
instance (Monad m, TermAlgebra m f) ⇒ TermMonad m f
```

Again, the free monad is the obvious instance of this typeclass.

```
instance Functor f ⇒ TermMonad (Free f) f
```

Its monadic primitives are implemented in terms of *fold*, *con* and *var*. In the abstract builder interface *TermMonad* we only partially expose this fact, by means of the following two laws. Firstly, the *var* operation should coincide with the monad's *return*.

$$var = return \tag{4}$$

Secondly, the monad's bind $(\gg\!=)$ should distribute through *con*.

$$con\ op \gg\!= k = con\ (fmap\ (\gg\!=k)\ op) \tag{5}$$

This law states that a term constructed by an operation, where the term is followed by a continuation k, is equivalent to constructing a term from an operation where the operation is followed by k. In other words, the arguments of an operation correspond to its continuations.

Examples All the compositional handlers of Section 2.2 can be easily expressed in terms of the more abstract *TermMonad* interface. For example, the revised nondeterminism handler looks as follows.

$$
\begin{array}{l}
\mathit{handle}'_{\mathit{Nondet}} :: \mathit{TermMonad}\ m\ g \Rightarrow \mathit{Free}\ (\mathit{Nondet} + g)\ a \to m\ [a] \\
\mathit{handle}'_{\mathit{Nondet}} = \mathit{fold}\ (\mathit{alg}'_{\mathit{Nondet}} \nabla \mathit{con})\ \mathit{gen}'_{\mathit{Nondet}} \\
\mathit{gen}'_{\mathit{Nondet}} :: \mathit{TermMonad}\ m\ g \Rightarrow a \to m\ [a] \\
\mathit{gen}'_{\mathit{Nondet}}\ x = \mathit{var}\ [x] \\
\mathit{alg}'_{\mathit{Nondet}} :: \mathit{TermMonad}\ m\ g \Rightarrow \mathit{Nondet}\ (m\ [a]) \to m\ [a] \\
\mathit{alg}'_{\mathit{Nondet}}\ (\mathit{Or}\ \mathit{ml}_1\ \mathit{ml}_2) = \\
\quad \mathbf{do}\ \{ l_1 \leftarrow \mathit{ml}_1\ ;\ l_2 \leftarrow \mathit{ml}_2\ ;\ \mathit{var}\ (l_1 +\!\!+\ l_2) \}
\end{array}
$$

Notice that not much change has been necessary. We have generalized away from *Free* g into a type m that is constrained by *TermMonad* $m\ g$.

3.2 Parametricity: Fold/Build Fusion for Free

Hinze et al. [6] state that we get fold/build fusion for free from the *free theorem* [22,21] of the builder's polymorphic type. Hence, let us consider what the new type of $\mathit{handler}_1$ buys us.

Theorem 1. *Assume that* F_1, F_2, H_1, G_1 *and* A *are closed types, with* F_1, F_2 *and* H_1 *functors. Given a function* h *of type* $\forall m\ .\ (\mathit{TermMonad}\ m\ F_2) \Rightarrow \mathit{Free}\ F_1\ A \to H_1\ (m\ (G_1\ A))$, *two term monads* M_1 *and* M_2 *and a term monad morphism* $\alpha :: \forall a\ .\ M_1\ a \to M_2\ a$, *then:*

$$\mathit{fmap}\ \alpha \cdot h_{M_1} = h_{M_2} \tag{6}$$

where the subscripts of h *denote the instantiations of the polymorphic type variable* m.

If $\mathit{handler}_2$ is a term monad morphism, then we can use the free theorem to determine $\mathit{pipeline}_{12}$ in one step, starting from Equation (3).

$$
\begin{array}{ll}
& \mathit{fmap}\ \mathit{handler}_2 \cdot \mathit{handler}_1 = \mathit{pipeline}_{12} \\
\equiv & \quad \{\ \text{Parametricity (6), assuming } \mathit{handler}_2 \text{ is a term monad morphism}\ \} \\
& \mathit{handler}_1 = \mathit{pipeline}_{12}
\end{array}
$$

Unfortunately, $\mathit{handler}_2$ is not a term monad morphism for the simple reason that H_2 is just an arbitrary functor that does not necessarily have a monad structure. Hence, in general H_2 is only term algebra.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{TermAlgebra}\ H_2\ F_2\ \mathbf{where} \\
\quad \mathit{var} = \mathit{gen}_2 \\
\quad \mathit{con} = \mathit{alg}_2
\end{array}
$$

Fortunately, we can turn any term algebra into a term monad, thanks to the codensity monad, which is what we explore in the next section.

3.3 Codensity: *TermMonad*s from *TermAlgebra*s

The Codensity Monad It is well-known that the codensity monad *Cod* turns any (endo)functor h into a monad (in fact, h need not even be a functor at all). It simply instantiates the generalised monoid of endomorphisms (e.g., see [16]) in the category of endofunctors.

```
newtype Cod h a = Cod {unCod :: ∀x . (a → h x) → h x}
instance Monad (Cod h) where
  return x = Cod (λk → k x)
  Cod m >>= f = Cod (λk → m (λa → unCod (f a) k))
```

TermMonad Construction Given any term algebra h for functor f, we have that *Cod* h is also a term algebra for f.

```
instance TermAlgebra h f ⇒ TermAlgebra (Cod h) f where
  var = return
  con = alg_Cod con
alg_Cod :: Functor f ⇒ (∀x . f (h x) → h x) → (f (Cod h a) → Cod h a)
alg_Cod alg op = Cod (λk → alg (fmap (λm → unCod m k) op))
```

Moreover, *Cod* h is also a term monad for f, even if h is not.

```
instance TermAlgebra h f ⇒ TermMonad (Cod h) f
```

The definition of *var* makes it easy to see that it satisfies the first term monad law in Equation (4). The proof for the second law, Equation (5) is less obvious:

```
  con op >>= f
≡   { unfold (>>=) }
  Cod (λk → unCod (con op) (λa → unCod (f a) k))
≡   { unfold con }
  Cod (λk → unCod (alg_Cod con op) (λa → unCod (f a) k))
≡   { unfold alg_Cod }
  Cod (λk → unCod (Cod (λk → con (fmap (λm → unCod m k) op))) (λa → unCod (f a) k))
≡   { apply unCod · Cod = id }
  Cod (λk → (λk → con (fmap (λm → unCod m k) op)) (λa → unCod (f a) k))
≡   { apply β-reduction }
  Cod (λk → con (fmap (λm → unCod m (λa → unCod (f a) k)) op))
≡   { apply β-expansion }
  Cod (λk → con (fmap (λm → (λk → unCod m (λa → unCod (f a) k)) k) op))
≡   { apply unCod · Cod = id }
  Cod (λk → con (fmap (λm → unCod (Cod (λk → unCod m (λa → unCod (f a) k))) k) op))
≡   { fold (>>=) }
  Cod (λk → con (fmap (λm → unCod (m >>= f) k) op))
≡   { apply β-expansion }
```

$$
\begin{array}{ll}
& \mathit{Cod}\ (\lambda k \to \mathit{con}\ (\mathit{fmap}\ (\lambda m \to (\lambda m \to \mathit{unCod}\ m\ k)\ (m \gg\!= k))\ \mathit{op})) \\
\equiv & \{\ \text{apply } \beta\text{-expansion}\ \} \\
& \mathit{Cod}\ (\lambda k \to \mathit{con}\ (\mathit{fmap}\ (\lambda m \to (\lambda m \to \mathit{unCod}\ m\ k)\ ((\lambda m \to m \gg\!= k)\ m))\ \mathit{op})) \\
\equiv & \{\ \text{fold } (\cdot)\ \} \\
& \mathit{Cod}\ (\lambda k \to \mathit{con}\ (\mathit{fmap}\ (\lambda m \to ((\lambda m \to \mathit{unCod}\ m\ k) \cdot (\lambda m \to m \gg\!= k))\ m)\ \mathit{op})) \\
\equiv & \{\ \text{apply } \eta\text{-reduction}\ \} \\
& \mathit{Cod}\ (\lambda k \to \mathit{con}\ (\mathit{fmap}\ ((\lambda m \to \mathit{unCod}\ m\ k) \cdot (\lambda m \to m \gg\!= k))\ \mathit{op})) \\
\equiv & \{\ \text{apply } \mathit{fmap}\text{-fission and unfold } (\cdot)\ \} \\
& \mathit{Cod}\ (\lambda k \to \mathit{con}\ (\mathit{fmap}\ (\lambda m \to \mathit{unCod}\ m\ k)\ (\mathit{fmap}\ (\lambda m \to m \gg\!= k)\ \mathit{op}))) \\
\equiv & \{\ \text{fold } \mathit{alg}_{\mathit{Cod}}\ \} \\
& \mathit{alg}_{\mathit{Cod}}\ \mathit{con}\ (\mathit{fmap}\ (\lambda m \to m \gg\!= k)\ \mathit{op}) \\
\equiv & \{\ \text{fold } \mathit{con} \text{ and } \eta\text{-reduce}\ \} \\
& \mathit{con}\ (\mathit{fmap}\ (\gg\!= f)\ \mathit{op})
\end{array}
$$

3.4 Shifting to Codensity

Now we can write $\mathit{handler}_2$ as the composition of a term monad morphism $\mathit{handler}'_2$ with a post-processing function $\mathit{runCod}\ \mathit{gen}_2$:

$$
\begin{array}{l}
\mathit{handler}_2 :: \mathit{Free}\ F_2\ a \to H_2\ a \\
\mathit{handler}_2 = \mathit{runCod}\ \mathit{gen}_2 \cdot \mathit{handler}'_2 \\
\mathit{handler}'_2 :: \mathit{Free}\ F_2\ a \to \mathit{Cod}\ H_2\ a \\
\mathit{handler}'_2 = \mathit{fold}\ (\mathit{alg}_{\mathit{Cod}}\ \mathit{alg}_2)\ \mathit{var} \\
\mathit{runCod} :: (a \to f\ x) \to \mathit{Cod}\ f\ a \to f\ x \\
\mathit{runCod}\ g\ m = \mathit{unCod}\ m\ g
\end{array}
$$

This decomposition of $\mathit{handler}_2$ hinges on the following property:

$$\mathit{fold}\ \mathit{alg}_2\ \mathit{gen}_2 = \mathit{runCod}\ \mathit{gen}_2 \cdot \mathit{fold}\ (\mathit{alg}_{\mathit{Cod}}\ \mathit{alg}_2)\ \mathit{var} \tag{7}$$

This equation follows from the second fusion law for folds (2), provided that:

$$
\begin{array}{r}
\mathit{gen}_2 = \mathit{runCod}\ \mathit{gen}_2 \cdot \mathit{var} \\
\mathit{runCod}\ \mathit{gen}_2 \cdot \mathit{alg}_{\mathit{Cod}}\ \mathit{alg}_2 = \mathit{alg}_2 \cdot \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)
\end{array}
$$

The former holds as follows:

$$
\begin{array}{ll}
& \mathit{runCod}\ \mathit{gen}_2 \cdot \mathit{var} \\
\equiv & \{\ \text{unfold } \cdot\ \} \\
& \lambda x \to \mathit{runCod}\ \mathit{gen}_2\ (\mathit{var}\ x) \\
\equiv & \{\ \text{unfold } \mathit{runCod} \text{ and } \mathit{var}\ \} \\
& \lambda x \to \mathit{unCod}\ (\mathit{Cod}\ (\lambda k \to k\ x))\ \mathit{gen}_2 \\
\equiv & \{\ \text{apply } \mathit{runCod} \cdot \mathit{Cod} = \mathit{id}\ \} \\
& \lambda x \to (\lambda k \to k\ x)\ \mathit{gen}_2 \\
\equiv & \{\ \beta\text{-reduction}\ \} \\
& \lambda x \to \mathit{gen}_2\ x
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\ \eta\text{-reduction}\ \} \\
& \mathit{gen}_2
\end{array}
$$

and the latter:

$$
\begin{array}{ll}
& \mathit{runCod}\ \mathit{gen}_2\ \cdot\ \mathit{alg}_{\mathit{Cod}}\ \mathit{alg}_2 \\
\equiv & \{\ \text{unfold}\ \cdot\ \} \\
& \lambda op \to \mathit{runCod}\ \mathit{gen}_2\ (\mathit{alg}_{\mathit{Cod}}\ \mathit{alg}_2\ op) \\
\equiv & \{\ \text{unfold}\ \mathit{runCod}\ \text{and}\ \mathit{alg}_{\mathit{Cod}}\ \} \\
& \lambda op \to \mathit{unCod}\ (\mathit{Cod}\ (\lambda k \to \mathit{alg}_2\ (\mathit{fmap}\ (\lambda m \to \mathit{unCod}\ m\ k)\ op)))\ \mathit{gen}_2 \\
\equiv & \{\ \text{apply}\ \mathit{runCod}\ \cdot\ \mathit{Cod} = \mathit{id}\ \} \\
& \lambda op \to (\lambda k \to \mathit{alg}_2\ (\mathit{fmap}\ (\lambda m \to \mathit{unCod}\ m\ k)\ op))\ \mathit{gen}_2 \\
\equiv & \{\ \beta\text{-reduction}\ \} \\
& \lambda op \to \mathit{alg}_2\ (\mathit{fmap}\ (\lambda m \to \mathit{unCod}\ m\ \mathit{gen}_2)\ op) \\
\equiv & \{\ \text{fold}\ \mathit{runCod}\ \} \\
& \lambda op \to \mathit{alg}_2\ (\mathit{fmap}\ (\lambda m \to \mathit{runCod}\ \mathit{gen}_2\ m)\ op) \\
\equiv & \{\ \eta\text{-reduction}\ \} \\
& \lambda op \to \mathit{alg}_2\ (\mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)\ op) \\
\equiv & \{\ \text{fold}\ \cdot\ \} \\
& \mathit{alg}_2\ \cdot\ \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)
\end{array}
$$

3.5 Fusion at Last

Finally, instead of fusing $\mathit{fmap}\ \mathit{handler}_2\ \cdot\ \mathit{handler}_1$ we can fuse $\mathit{fmap}\ \mathit{handler}'_2\ \cdot\ \mathit{handler}_1$ using the free theorem. This yields:

$$\mathit{pipeline}'_{12} = \mathit{fold}\ \mathit{alg}_1\ \mathit{gen}_1$$

Now we can calculate the original fusion:

$$
\begin{array}{ll}
& \mathit{fmap}\ \mathit{handler}_2\ \cdot\ \mathit{handler}_1 \\
\equiv & \{\ \text{decomposition of}\ \mathit{handler}_2\ \} \\
& \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2\ \cdot\ \mathit{handler}'_2)\ \cdot\ \mathit{handler}_1 \\
\equiv & \{\ \mathit{fmap}\ \text{fission}\ \} \\
& \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)\ \cdot\ \mathit{fmap}\ \mathit{handler}'_2\ \cdot\ \mathit{handler}_1 \\
\equiv & \{\ \text{free theorem}\ \} \\
& \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)\ \cdot\ \mathit{handler}_1
\end{array}
$$

In other words, the fused version can be defined as:

$$\mathit{pipeline}_{12} = \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)\ \cdot\ \mathit{fold}\ \mathit{alg}_1\ \mathit{gen}_1$$

Observe that this version only performs a single *fold* and does not allocate any intermediate tree.

3.6 Repeated Fusion

Often a sequence of handlers is not restricted to two. Fortunately, we can easily generalize the above to a pipeline of n handlers

$$\mathit{fmap}^n\ \mathit{handler}_n \cdot \ldots \cdot \mathit{fmap}\ \mathit{handler}_1 \cdot \mathit{handler}_0$$

where $\mathit{handler}_i = \mathit{fold}\ \mathit{alg}_i\ \mathit{gen}_i$ $(i \in 1..n)$. This pipeline fusion can start by arbitrarily fusing two consecutive handlers $\mathit{handler}_j$ and $\mathit{handler}_{j+1}$ using the above approach, and then incrementally extending the fused kernel on the left and the right with additional handlers. These two kinds of extensions are explained below.

Fusion on the Right Suppose that $\mathit{fmap}\ \mathit{handler}_2 \cdot \mathit{handler}_1$ is composed with another handler on the right:

$$\begin{aligned}&\mathit{handler}_0 :: (\mathit{TermMonad}\ m_1\ F_1) \Rightarrow \mathit{Free}\ F_0\ a \to H_0\ (m_1\ (G_0\ a))\\&\mathit{handler}_0 = \mathit{fold}\ \mathit{alg}_0\ \mathit{gen}_0\end{aligned}$$

to form the pipeline:

$$\mathit{pipeline}_{012} = \mathit{fmap}\ (\mathit{fmap}\ \mathit{handler}_2 \cdot \mathit{handler}_1) \cdot \mathit{handler}_0$$

Can we perform the fusion twice to obtain a single *fold* and eliminate both intermediate trees? Yes! The first fusion, as before yields:

$$\mathit{pipeline}_{012} = \mathit{fmap}\ (\mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2) \cdot \mathit{handler}_1) \cdot \mathit{handler}_0$$

Applying *fmap* fission and regrouping, we obtain:

$$\mathit{pipeline}_{012} = \mathit{fmap}\ (\mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)) \cdot (\mathit{fmap}\ \mathit{handler}_1 \cdot \mathit{handler}_0)$$

Now the right component is another instance of the binary fusion problem, which yields:

$$\mathit{pipeline}_{012} = \mathit{fmap}\ (\mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_2)) \cdot \mathit{fmap}\ (\mathit{runCod}\ \mathit{gen}_1) \cdot \mathit{handler}_0$$

Fusion on the Left Suppose that $\mathit{handler}_2$ has the more specialised type:

$$\mathit{handler}_2 :: (\mathit{TermMonad}\ m_3\ F_3) \Rightarrow \mathit{Free}\ F_2\ a \to H_2\ (m_2\ (G_2\ a))$$

then we can compose $\mathit{fmap}\ \mathit{handler}_2 \cdot \mathit{handler}_1$ on the left with another handler:

$$\begin{aligned}&\mathit{handler}_3 :: \mathit{Free}\ F_3\ a \to H_3\ a\\&\mathit{handler}_3 = \mathit{fold}\ \mathit{alg}_3\ \mathit{gen}_3\end{aligned}$$

This yields a slightly more complicated fusion scenario:

$$pipeline_{123} = fmap\ (fmap\ handler_3 \cdot handler_2) \cdot handler_1$$

Of course, we can first fuse $handler_2$ and $handler_3$. That would yield an instance of fusion on the right. However, suppose we first fuse $handler_1$ and $handler_2$, after applying *fmap* fission.

$$pipeline_{123} = fmap\ (fmap\ handler_3) \cdot fmap\ (runCod\ gen_2) \cdot handler_1$$

Now we can shift the carrier of $handler_3$ to codensity and invoke the free theorem on $fmap\ (runCod\ gen_2) \cdot handler_1$. This accomplishes the second fusion.

$$pipeline_{123} = fmap\ (fmap\ (runCod\ gen_3)) \cdot fmap\ (runCod\ gen_2) \cdot handler_1$$

Summary An arbitrary pipeline of the form:

$$fmap^n\ handler_n \cdot \ldots \cdot fmap\ handler_1 \cdot handler_0$$

where $handler_i = fold\ alg_i\ gen_i\ (i \in 1..n)$ fuses into

$$fmap^n\ (runCod\ g_n) \cdot \ldots \cdot fmap\ (runCod\ gen_1) \cdot handler_0$$

3.7 Fusion all the Way

We are not restricted to fusing handlers, but can fuse all the way, up to and including the expression that builds the initial AST and to which the handlers are applied. Consider for example the *coin* example of Section 2.1. The free theorem of *coin*'s type is a variant of Theorem 1:

$$\alpha\ coin = coin$$

where $\alpha :: \forall a\ .\ M_1\ a \to M_2\ a$ is a term monad morphism between any two term monads M_1 and M_2. We can use this to fuse $handle_{Nondet}\ coin$ into $runCod\ gen_{Nondet}\ coin$. Of course this fusion interacts nicely with the fusion of a pipeline of handlers.

4 Pragmatic Implementation and Evaluation

This section turns the fusion approach into a practical Haskell implementation and evaluates the performance improvement.

4.1 Pragmatic Implementation

Before we can put the fusion approach into practice, we need to consider a few pragmatic implementation aspects.

Inlining with Typeclasses In a lazy language like Haskell, fusion only leads to a significant performance gain if it is performed statically by the compiler and combined with inlining. In the context of the GHC compiler, the inlining requirement leaves little implementation freedom: GHC is rather reluctant to inline in general recursive code. There is only one exception: GHC is keen to create type-specialised copies of (constraint) polymorphic recursive definitions and to inline the definitions of typeclass methods in the process.

In short, if we wish to get good speed-ups from effect handler fusion, we need to make sure that the effectful programs are polymorphic in the term monad and that all the algebras are held in typeclass instances. For this reason, all handlers should be made instances of *TermAlgebra*.

Explicit Carrier Functors The carrier functor of the compositional state handler is $s \to m_2-$. From the category theory point of view, this is clearly a functor. However, it is neither an instance of the Haskell *Functor* typeclass nor can it be made one in this syntactically invalid form. A new type needs to be created that instantiates the functor typeclass:

```
newtype StateCarrier s m a = SC {unSC :: s → m a}
instance Functor m ⇒ Functor (StateCarrier s m) where
  fmap f x = SC (fmap (fmap f) (unSC x))
instance TermMonad m₂ f
  ⇒     TermAlgebra (StateCarrier s m₂) (State s + f) where
  con =  SC · (alg'_State ∇ con_State) · fmap unSC
  var =  SC · gen'_State
gen'_State :: TermMonad m f ⇒ a → (s → m a)
gen'_State x = const (var x)
alg'_State :: TermMonad m f ⇒ State s (s → m a) → (s → m a)
alg'_State (Put s' k) s =     k s'
alg'_State (Get k)    s =     k s s
```

Now the following function is convenient to run a fused state handler.

```
runStateC :: TermMonad m₂ f ⇒ Cod (StateCarrier s m₂) a → (s → m₂ a)
runStateC = unSC · runCod var
```

Unique Carrier Functors Even though the logging state handler has ostensibly[4] the same carrier $s \to m_2-$ as the regular compositional state handler, we cannot reuse the same functor. The reason is that in the typeclass-based approach the carrier functor must uniquely determine the algebra; two different typeclass instances for the same type are forbidden. Hence, we need to write a new set of definitions as follows:

```
newtype LogStateCarrier s m a = LSC {unLSC :: s → m a}
```

[4] The typeclass constraints on m_2 are different.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ m \Rightarrow \mathit{Functor}\ (\mathit{LogStateCarrier}\ s\ m)\ \textbf{where} \\
\quad \mathit{fmap}\ f\ x = \mathit{LSC}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ (\mathit{unLSC}\ x)) \\
\textbf{instance}\ \mathit{TermMonad}\ m_2\ (\mathit{Writer}\ \mathit{String} + \mathit{Void}) \\
\quad \Rightarrow \mathit{TermAlgebra}\ (\mathit{LogStateCarrier}\ s\ m_2)\ (\mathit{State}\ s)\ \textbf{where} \\
\quad \mathit{var} = \mathit{LSC} \cdot \mathit{gen'}_{\mathit{State}} \\
\quad \mathit{con} = \mathit{LSC} \cdot \mathit{alg}_{\mathit{LogState}} \cdot \mathit{fmap}\ \mathit{unLSC} \\
\mathit{runLogStateC} :: \mathit{TermMonad}\ m_2\ (\mathit{Writer}\ \mathit{String} + \mathit{Void}) \\
\qquad \Rightarrow \mathit{Cod}\ (\mathit{LogStateCarrier}\ s\ m_2)\ a \to (s \to m_2\ a) \\
\mathit{runLogStateC} = \mathit{unLSC} \cdot \mathit{runCod}\ \mathit{var}
\end{array}
$$

4.2 Evaluation

To evaluate the impact of fusion we consider several benchmarks implemented in different ways: running handlers over the inductive definition of the free monad (FREE) and to its Church encoding (CHURCH), the fully fused definitions (FUSED), and the conventional definitions of the state monad from MTL (MTL).

The benchmarks are run in the Criterion benchmarking harness using GHC 7.10.1 on a MacBook Pro with a 3 GHz Intel Core i7 processor, 16 GB RAM and Mac OS 10.10.3. All values are in milliseconds, and show the ordinary least-squares regression of the recorded samples; the R^2 goodness of fit is above 0.99 in all instances.

Benchmark	FREE	CHURCH	FUSED	MTL
count$_1$				
10^7	1,017	1,311	3	3
10^8	10,250	13,220	29	29
10^9	103,000	129,700	291	295
count$_2$				
10^6	684	746	167	213
10^7	6,937	7,344	1,740	2,157
10^8	102,700	98,010	17,220	20,300
count$_3$				
10^6	559	555	166	205
10^7	5,759	5,561	1,618	2,132
10^8	110,900	94,760	16,300	20,120
grammar	794	763	6	77
pipes	1,325	1,351	43	N/A

The *count*$_1$ benchmark consists of the simple count-down loop used by Kammar et al. [9].

$count_1 =$
  do $i \leftarrow get$
    if $i \equiv 0$ **then** $return\ i$
      else $put\ (i - 1)\ ;\ count_1$

We have evaluated this program with three different initial states of 10^7, 10^8 and 10^9. The results show that all representations scale linearly with the size of the benchmark. However, the fused representation is about 300 times faster than the free monad representations and matches the performance of the traditional monad transformers.

The $count_2$ benchmarks extends the $count_1$ benchmark with a *tell* operation from the *Writer* effect in every iteration of the loop. It is run by sequencing the state and writer handler. The improvement due to fusion is now much less extreme, but still quite significant.

The $count_3$ benchmark is the $count_1$ program, but run with the logging state handler that delegates to the writer handler. The runtimes are slightly better than those of $count_2$.

The *grammar* benchmark implements a simpler parser by layering the state and non-determinism effects. Again fusion has a tremendous impact, even considerably outperforming the MTL implementation.

The *pipes* benchmark consists of the simple producer-consumer pipe used by Kammar et al. [9]. We can see that fusion provides a significant improvement over either free monad representation. There is no sensible MTL implementation to compare with for this benchmark.

The results (in ms) show that the naive approaches based on intermediate trees, either defined inductively or by Church encoding, incur a considerable overhead compared to traditional monads and monad transformers. Yet, thanks to fusion they can easily compete or even slightly outperform the latter.

5 Related Work

5.1 Fusion

Fusion has received much attention, and was first considered as the elimination of trees by Wadler [23] with his so-called deforestation algorithm, which was then later generalized by Chin [3].

From the implementation perspective, Gill et al. first introduced the notion of shortcut fusion to Haskell [4], thus allowing programs written as folds over lists to be fused. Takano and Meijer showed how fusion could be generalized to arbitrary datastructures [19]. The technique of using free theorems to explain certain kinds of fusion was applied by Voigtländer [20].

Work by Hinze et al. [6] builds on recursive coalgebras to show the theory and practice of fusion, limited to the case of folds and unfolds. Later work by Harper [5] provides a pragmatic discussion, that bridges the gap between theory and practice further, by discussing the implementation of fusion in GHC with inline rules. Harper also considers the fusion of Church encodings.

More recently, recursive coalgebras appear in work by Hinze et al. [7] where conjugate hylomorphisms are introduced as a means of unifying all presently known structured recursion schemes. There, the theory behind fusion for general datatypes across all known schemes is described as arising from naturality laws of adjunctions. Special attention is drawn to fusion of the cofree comonad, which is the dual case of free monads we consider here.

5.2 Effect Handlers

Plotkin and Power were the first to explore effect operations [14], and gave an algebraic account of effects [15] and their combination [8]. Subsequently, Plotkin and Pretnar [13] have added the concept of handlers to deal with exceptions. This has led to many implementation approaches.

Lazy Languages Many implementations of effect handlers focus on the lazy language Haskell.

For the sake of simplicity and without regard for efficiency, Wu et al. [24] use the inductive datatype representation of the free monad for their exposition. They use the *Data Types à la Carte* approach [18] to conveniently inject functors into a co-product; that approach is entirely compatible with this paper. Wu et al. also generalize the free monad to *higher-order functors*; we expect that our fusion approach generalizes to that setting.

Kiselyov et al. [10] provide a Haskell implementation in terms of the free monad too. However, they combine this representation with two optimizations: 1) the codensity monad improves the performance of ($\gg=$), and 2) their *Dynamic*-based open unions have a better time complexity than nested co-products. Due to the use of the codensity monad, this paper also benefits from the former improvement. Moreover, we believe that the latter improvement is unnecessary due to the specialisation and inlining opportunities that are exposed by fusion.

Van der Ploeg and Kiselyov [12] present an implementation of the free monad with good asymptotic complexity for both pattern matching and binding; unfortunately, the constant factors involved are rather high. This representation is mainly useful for effect handlers that cannot be easily expressed as folds, and thus fall outside of the scope of the current paper.

Behind a Template Haskell frontend Kammar et al. [9] consider a range of different Haskell implementations and perform a performance comparison. Their basic representation is the inductive datatype definition, with a minor twist: the functor is split into syntax for the operation itself and a separate continuation. This representation is improved with the codensity monad. Finally, they provide—without explanation—a representation that is very close to the one presented here; their use of the continuation monad instead of the codensity monad is a minor difference.

Both Atkey et al. [1] and Schrijvers et al. [17] study the interleaving of a free monad with an arbitrary monad, i.e., the combination of algebraic effect handlers and conventional monadic effects. We believe that our fusion technique can be adapted for optimizing the free monad aspect of their settings.

Strict Languages In the absence of lazy evaluation, the inductive datatype definition of the free monad is not practical.

Kammar et al. [9] briefly sketch an implementation based on delimited continuations. Schrijvers et al. [17] show the equivalence between a delimited continuations approach and the inductive datatype; hence the fusion technique presented in this paper is in principle possible. However, in practice, the codensity monad used for fusion is likely not efficient in strict languages. Hence, effective fusion for strict languages remains to be investigated.

5.3 Monad Transformers

Monad transformers, as first introduced by Liang et al. [11], pre-date algebraic effect handlers as a means for modelling compositional effects. Yet, there exists a close connection between both approaches: monad transformers are fused forms of effect handlers. What is particular about their underlying effect handlers is that their carriers are term algebras with a monadic structure, i.e., term monads. This means that the *Cod* construction is not necessary for fusion.

6 Conclusion

We have explained how to fuse algebraic effect handlers by shifting perspective from free monads to term monads. Our benchmarks show that, with a careful code setup in Haskell, this leads to good speed-ups compared to the free monad, and allows algebraic effect handlers to compete with the traditional monad transformers.

Acknowledgments

The authors would like to thank Ralf Hinze for suggesting that they consider the specification that free monads satisfy in terms of adjunctions, and to James McKinna who helped baptize *con* for constructing terms. We are also grateful for the feedback from the members of the IFIP Working Group 2.1 on an early iteration of this work. This work has been partially funded by the Flemish Fund for Scientific Research (FWO).

References

1. Atkey, R., Johann, P., Ghani, N., Jacobs, B.: Interleaving data and effects (2012), submitted for publication
2. Burris, S., Sankappanavar, H.P.: A Course in Universal Algebra. No. 78 in Graduate Texts in Mathematics, Springer-Verlag (1981)
3. Chin, W.N.: Safe fusion of functional expressions ii: Further improvements. Journal of Functional Programming 4, 515–555 (Oct 1994)
4. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. pp. 223–232. FPCA '93, ACM, New York, NY, USA (1993)

5. Harper, T.: A library writer's guide to shortcut fusion. In: Proceedings of the 4th ACM Symposium on Haskell. pp. 47–58. Haskell '11, ACM, New York, NY, USA (2011)
6. Hinze, R., Harper, T., James, D.: Theory and practice of fusion. In: Hage, J., Morazán, M. (eds.) Implementation and Application of Functional Languages, Lecture Notes in Computer Science, vol. 6647, pp. 19–37. Springer Berlin Heidelberg (2011)
7. Hinze, R., Wu, N., Gibbons, J.: Conjugate hylomorphisms – the mother of all structured recursion schemes. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 527–538. POPL '15, ACM, New York, NY, USA (2015)
8. Hyland, M., Plotkin, G.D., Power, J.: Combining effects: Sum and tensor. Theoretical Computer Science 357(1-3), 70–99 (2006)
9. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming. pp. 145–158. ICFP '14, ACM (2013)
10. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: an alternative to monad transformers. In: Proceedings of the 2013 ACM SIGPLAN symposium on Haskell. pp. 59–70. Haskell '13, ACM (2013)
11. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 333–343. POPL '95, ACM, New York, NY, USA (1995)
12. Ploeg, A.v.d., Kiselyov, O.: Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 133–144. Haskell '14, ACM, New York, NY, USA (2014)
13. Plotkin, G.D., Matija, P.: Handling algebraic effects. Logical Methods in Computer Science 9(4) (2013)
14. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 2303, pp. 342–356. Springer (2002)
15. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Applied Categorical Structures 11(1), 69–94 (2003)
16. Rivas, E., Jaskelioff, M.: Notions of computation as monoids. CoRR abs/1406.4823 (2014), `http://arxiv.org/abs/1406.4823`
17. Schrijvers, T., Wu, N., Desouter, B., Demoen, B.: Heuristics entwined with handlers combined. In: Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming. pp. 259–270. PPDP '14, ACM (Sep 2014)
18. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008)
19. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 306–313. FPCA '95, ACM, New York, NY, USA (1995)
20. Voigtländer, J.: Proving correctness via free theorems: The case of the destroy/build-rule. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. pp. 13–20. PEPM '08, ACM, New York, NY, USA (2008)
21. Voigtländer, J.: Free theorems involving type constructor classes: Functional pearl. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 173–184. ICFP '09, ACM, New York, NY, USA (2009)

22. Wadler, P.: Theorems for free! In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 347–359. FPCA '89, ACM, New York, NY, USA (1989)
23. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science 73(2), 231–248 (Jan 1990)
24. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 1–12. Haskell '14, ACM, New York, NY, USA (2014)