Modular Components with Monadic Effects

Tom Schrijvers[1] and Bruno C. d. S. Oliveira[2]

[1] Katholieke Universiteit Leuven
tom.schrijvers@cs.kuleuven.be
[2] ROSAEC Center, Seoul National University
bruno@ropas.snu.ac.kr

Abstract. Limitations of monad stacks get in the way of developing highly modular programs with effects. This paper demonstrates that Functional Programming's abstraction tools are up to the challenge. Of course, abstraction must be followed by clever instantiation: Huet's zipper for the monad stack makes components jump through unanticipated hoops.

1 Introduction

Features like higher-order functions and parametric polymorphism make functional languages very suitable for developing highly modular code (libraries or components) that can be reused in many different settings.

Purely functional languages like Haskell, also provide good means to reason about (modular) code, for instance through equational reasoning and parametricity. Purity bans implicit side-effects because it destroys the reasoning capabilities. Instead, monads are the preferred way for introducing explicit effects, without losing equational reasoning and parametricity.

Unfortunately, the combination of modularity and monads is currently the Achilles heel of the purely functional approach. Monads seem to be subpar to imperative object- and aspect-oriented languages for highly modular and effectful applications. In fact, functional programmers do not easily explore such applications, because working with monads quickly becomes "awkward".

Monad transformers are the most prominent work on modular monads. However, they only provide limited modularity. Only components with distinct effects can be composed without problems. This imposes a serious restriction on the number of effectful components that can be composed while reusing existing monad transformer implementations.

In this article we provide solutions to the construction of highly modular software with monadic effects. Our approach does not impose restrictions on the number of components or the reuse of existing monad implementations. It neither requires the component implementation to anticipate particular system configurations nor does it restrict the eventual configurations.

2 Setting

To illustrate our approach, we consider a setting that provides modular effects through monad transformers and modular components through open recursion:

- We use the monad transformer variant defined by the Monatron library (1) that provides an essential operation on monad transformer stacks. This operation can be defined for other transformer libraries too, but would have to be done from scratch.
- We use the mixin approach to open recursion pioneered by Cook (2), because it provides a minimal yet highly expressive approach (e.g. in terms of control flow) to tightly coupled components. Alternative approaches in terms of type classes and imposed control flow patterns are possible too.

2.1 Monad Transformers in Monatron

We assume that the reader is already familiar with the general concept of monad transformers, and summarize here only the particulars of Monatron. The type class $MonadT\ t$ expresses that t is a monad transformers. This means that t provides the usual lifting functionality $lift :: m\ a \to t\ m\ a$. Furthermore, it also supplies a bind operator $tbind :: t\ m\ a \to (a \to t\ m\ b) \to t\ m\ b$ for the transformed monad; and $treturn :: a \to t\ m\ a$, which is simply defined by default as $lift \circ return$. A distinguishing feature of Monatron's monad transformers is the *tmixmap* method:

$$\begin{aligned} tmixmap\ &:: (Monad\ m, Monad\ n) \\ &\Rightarrow (forall\ a \circ m\ a \to n\ a) \\ &\to (forall\ b \circ n\ b \to\ m\ b) \to t\ m\ c \to t\ n\ c \end{aligned}$$

The *tmixmap* operation takes a natural isomorphism—two natural transformations, from the monad functor m to the monad functor n and vice-versa, that are each other's inverse—and returns a natural transformation from the monad functor $t\ m$ to the monad functor $t\ n$. In other words, *tmixmap* is an operation similar to the *fmap* operation of the *Functor* class, but mapping the monad functor instead. However, unlike *fmap* the (higher-order) functor t can have both co-variant and contra-variant occurrences (for the continuation monad transformer in particular) of m, which explains the need for the natural isomorphism.

Monad Transformer Implementations Monatron provides the usual range of monad transformer implementations, summarized in Figure 1.

As an example of the underlying implementation approach, consider the state monad with its methods for reading and writing the state. Monatron provides an explicit *dictionary* type

type *MakeWith s m*

that encapsulates the functionality for accessing a state of type s in monad m. The actual methods can be retrieved from this dictionary with helper functions:

```
  -- identity monad
newtype 𝕀 a
𝕀      :: a → 𝕀 a
run𝕀 :: 𝕀 a → a

  -- identity transformer
newtype 𝕀T m a
𝕀T     :: m a → 𝕀T m a
run𝕀T :: 𝕀T m a → m a

  -- reader transformer
newtype ℝT e m a
ℝT     :: (e → m a) → ℝT e m a
runℝT :: e → ℝT e m a → m a

  -- reader class
class Monad m ⇒ ℝM e m | m → e
ask    :: ℝM e m ⇒ m e

  -- state transformer
newtype 𝕊T s m a
𝕊T     :: (s → m (a, s)) → 𝕊T s m a
run𝕊T :: s → 𝕊T s m a → m (a, s)

  -- state class
class Monad m ⇒ 𝕊M s m | m → s
get    :: 𝕊M s m ⇒ m s
put    :: 𝕊M s m ⇒ s → m ()

  -- exception transformer
newtype 𝔼T x m a
𝔼T     :: m (Either x a) → 𝔼T x m a
run𝔼T :: 𝔼T x m a → m (Either x a)

  -- exception class
class Monad m ⇒ 𝔼M x m | m → x
throw :: 𝔼M x m ⇒ x → m a
```

Fig. 1. Monatron quick reference.

```
getX :: Monad m ⇒ MakeWith s m → m s
putX :: Monad m ⇒ MakeWith s m → s → m ()
```

In addition, the state functionality can be lifted through other monad transformers that reside above the state transformer in the monad stack. The lifting is uniform: there is a single implementation for lifting the state transformer methods through all other monad transformers:

```
liftMakeWith :: (Monad m, MonadT t) ⇒
                MakeWith z m → MakeWith z (t m)
```

For convenience Monatron uses Haskell's type class mechanism to make the dictionaries implicit. Figure 2 shows how this is done for the state monad. The first instance implements the specific functionality for the $\mathbb{S}_T$ monad using *makeWithStateT*. The second instance is more interesting as it shows how Monatron makes use of *liftMakeWith* to achieve uniform lifting through *any* monad transformer t. Note that other monad transformers are implemented in essentially the same way: one instance provides the functionality specific to the particular monad in question; whereas another instance provides uniform lifting.

```
class Monad m ⇒ 𝕊M z m | m → z where
  stateM :: MakeWith z m
instance Monad m ⇒ 𝕊M z (𝕊T z m) where
  stateM = makeWithStateT
instance (𝕊M z m, MonadT t) ⇒ 𝕊M z (t m) where
  stateM = liftMakeWith stateM
get :: 𝕊M z m ⇒ m z
get = getX stateM
put :: 𝕊M z m ⇒ z → m ()
put = putX stateM
```

Fig. 2. Overloading of state operations in Monatron

2.2 Mixins and Open Recursion

We briefly summarize the notion of mixins, and refer the interested reader to previous literature on the topic for a more indepth treatment. The basis of our mixin implementation in Haskell is:

```
type Mixin s = s → s
fix :: Mixin s → s
fix a = a (fix a)
(⊛) :: Mixin s → Mixin s → Mixin s
a1 ⊛ a2 = λproceed → a1 (a2 proceed)
```

The type *Mixin s* is a synonym for a function with type $s \to s$ representing open recursion. The parameter of that function is called a *join point*, that is, the point in the component at which another component is added. The operation $\circledast$ defines component composition. The function *fix* is a fixpoint combinator used for closing, or sealing, an open and potentially composed component.

Consider the following open functions:

```
fib1 :: Mixin (Int → Int)
fib1 proceed n = case n of
  0 → 0
  1 → 1
```

```
advfib :: Mixin (Int → Int)
advfib proceed n = case n of
  10 → 55
  _  → proceed n
```

The open function fib_1 defines the standard fibonacci function, except that recursive calls are replaced by *proceed*. The open function *advfib* optimizes one call of the fibonacci function by returning the appropriate value immediately. Different combinations of open functions are closed through *fix*:

```
slowfib1 , optfib :: Int → Int
slowfib1 = fix fib1
optfib   = fix (advfib ⊛ fib1 )
```

2.3 Effectful Components

Components with effects are obtained by combining mixins and monads. For instance, consider the following memoization component and a monadic fibonacci function.

```
memo :: 𝕊M (Map Int Int) m ⇒ Mixin (Int → m Int)
memo proceed x =
  do m ← get
     if member x m then return (m ! x)
                   else do y ← proceed x
                           m′ ← get
                           put (insert x y m′)
                           return y
fib2 :: Monad m ⇒ Mixin (Int → m Int)
fib2 proceed n = case n of
  0 → return 0
  1 → return 1
  _ → do y ← proceed (n − 1)
         x ← proceed (n − 2)
         return (x + y)
```

We can instantiate different monads, using the corresponding run functions of Figure 1, to recover variations of the fibonacci function. For example, the identity monad recovers the effect-free function

```
slowfib2 :: Int → Int
slowfib2 = run𝕀 ∘ fix fib2
```

while a fast fibonacci function is obtained by adding the memo advice and suitably instantiating the state monad:

```
fastfib :: Int → Int
fastfib n = run𝕀 ∘ eval𝕊T empty $ fix (memo ⊛ fib2) n
```

where

```
eval𝕊T :: Monad m ⇒ s → 𝕊T s m a → m a
eval𝕊T s m = run𝕊T s m ≫= return ∘ fst
```

Another component for profiling is

```
prof :: 𝕊M Int m ⇒ Mixin (a → m b)
prof proceed x =
  do c ← get
     put (c + 1)
     proceed x
```

which allows us to count the number of calls to the fibonacci function

```
proffib n = run𝕀 ∘ eval𝕊T 0 $ fix (prof ⊛ fib2) n
```

Transformer Conflicts Of course, we would also like to profile the memoized fibonacci function to get an idea of how much more efficient it is.

$$\begin{aligned}&\mathit{profmemofib} :: \mathit{Int} \to \mathbb{S}_{\mathrm{T}}\ \mathit{Int}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ \mathbb{I})\ \mathit{Int}\\&\mathit{profmemofib} = \mathit{fix}\ (\mathit{prof} \circledast \mathit{memo} \circledast \mathit{fib}_2)\end{aligned}$$

Unfortunately, the type checker complains that *Int* and *Map Int Int* are distinct types. The problem is that there are two uses of *get* in our features: one in *evalMem*; and another in *evalVar*. Due to automatic lifting, both *get* methods read the state from the same top-level $\mathbb{S}_{\mathrm{T}}$, which happens to contain an *Int* value. This is the right thing to do for *prof*, but wrong for *memo* that expects a value of type *Map Int Int*.

This type error is only a symptom of the real problem though. Namely, we expect the *get* calls in *prof* and *memo* to pick, or automatically lift out, different states in the monad stack, but the type checker does not distinguish between the two calls.

The lifting is biased towards the top of the monad stack. If the stack contains two $\mathbb{S}_{\mathrm{T}}$ instances, then the top one is in focus. Obviously, we cannot simply rearrange the layers in the monad stack to fix the problem, because this also alters the semantics and still the bottom instance remains inaccessible.

In Liang et al.'s modular interpreters this problem is solved using *lift* methods to explicitly disambiguate the access to the monad stack. However, this solution would not work for us because, unlike Liang et al., we are interested in having modular components that can be reused in several different configurations; and where potentially many different interpreters can coexist at the same time. The use of *lift* entails adapting existing code for the library components, which is fine when a single instance of a modular interpreter is in use, but it leads to fundamentally incompatible components when multiple interpreters with different configurations exist.

A dilemma At this stage it seems that we are left with a dilemma. On the one hand automatically lifted methods like *get* are nice because they do not pollute the code and they interact well with abstraction, implicitly lifting the monad into the right layer. Unfortunately, they do not allow multiple instances of the same monad in the monad stack, which is just too constraining for realistic applications. On the other hand explicit *lift* methods are nice to disambiguate uses of automatically lifted methods, which allows multiple monads of the same type to be used in a component. However *lift* methods can also lead to a significant loss of abstraction and reuse.

Is there a way out of this dilemma?

3 The Monad Zipper

We want to combine multiple instances of the same monad without touching the library components. In order to have our cake and eat it too, we must make the most of the provided abstraction. Indeed, we can influence the behavior

of the library components from the outside by instantiating the type variables appropriately. Of course, doing so in the obvious way did not get us anywhere earlier. So we need to reconsider what we expect from the instantiation: it should focus the automatic lifting to the desired layer in the stack.

3.1 Stacks and Zippers

Sometimes type-level problems get easier when we shift them to the value level. Let's reify the structure of the monad stack in a data type

$$\begin{aligned}&\textbf{data}\ \mathit{Stack} &&= \mathit{Push}\ \mathit{Trans}\ \mathit{Stack} \mid \mathit{Bottom}\ \mathit{Monad}\\ &\textbf{data}\ \mathit{Trans} &&= T_1 \mid \ldots \mid T_n\\ &\textbf{data}\ \mathit{Monad} &&= M_1 \mid \ldots \mid M_n\end{aligned}$$

where the T_i represent the different transformers and M_i are plain monads like $\mathbb{I}$.

(3) taught us how to shift the focus to any position in a data structure, with his *zipper*. Here is the *Zipper* for *Stack*:

$$\begin{aligned}&\textbf{data}\ \mathit{Zipper} &&= \mathit{Zipper}\ \mathit{Path}\ \mathit{Trans}\ \mathit{Stack}\\ &\textbf{data}\ \mathit{Path} &&= \mathit{Pop}\ \mathit{Trans}\ \mathit{Path} \mid \mathit{Top}\end{aligned}$$

where $\mathit{Zipper}\ p\ l\ s$ denotes a stack with layer l in focus, remainder of the stack s and path p back to the top of the stack. The path is a reversed list, where the first element is closest to the layer in focus and the last element is the top of the stack.

The *zipper* function turns a stack into a zipper with the first element in focus:

$$\begin{aligned}&\mathit{zipper} :: \mathit{Stack} \to \mathit{Zipper}\\ &\mathit{zipper}\ (\mathit{Push}\ t\ s) = \mathit{Zipper}\ \mathit{Top}\ t\ s\end{aligned}$$

while the *up* and *down* functions allow shifting the focus one position up or down:

$$\begin{aligned}&\mathit{up}, \mathit{down} :: \mathit{Zipper} \to \mathit{Zipper}\\ &\mathit{up}\ (\mathit{Zipper}\ (\mathit{Pop}\ t_1\ p)\ t_2\ s) &&= \mathit{Zipper}\ p\ t_1\ (\mathit{Push}\ t_2\ s)\\ &\mathit{down}\ (\mathit{Zipper}\ p\ t_1\ (\mathit{Push}\ t_2\ s)) &&= \mathit{Zipper}\ (\mathit{Pop}\ t_1\ p)\ t_2\ s\end{aligned}$$

It's all well and good to zip around a reified form of the monad stack, but can we do it on the real thing too?

3.2 Monad Zipper

The answer is yes. Here is how the monad zipper ($\triangleright$) is defined:

$$\textbf{newtype}\ (t_1 \triangleright t_2)\ m\ a = \mathbb{Z}_{\mathrm{T}}\ \{\mathit{run}\mathbb{Z}_{\mathrm{T}} :: t_1\ (t_2\ m)\ a\}$$

where the type $(p \triangleright t)\ s$ corresponds to the reified data structure $\mathit{Zipper}\ p\ t\ s$. However, the monad zipper only changes the type representation: the newtype

indicates that no actual structural change to the monad stack $t_1\ (t_2\ m)$ takes place. Even though the zipper shifts the focus to t_2, it does keep t_1 around, which is essential for recursive calls that need to shift the focus back to t_1. Such recursive calls are impossible using *lift* because the information about t_1 is lost in lifted code.

Mixing Stack and Zipper The type system will not allow values of type *Zipper* to be used when values of type *Stack* are expected. This segregation is not the case at the type level: the monad zipper type ($\triangleright$) can appear as part of a monad stack. Indeed, we define $t_1 \triangleright t_2$ to be a monad transformer that is the composition of t_1 and t_2:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{MonadT}\ t_1, \mathit{MonadT}\ t_2)\\
&\quad \Rightarrow \mathit{MonadT}\ (t_1 \triangleright t_2)\ \textbf{where}\\
&\qquad \mathit{lift} \qquad\quad = \mathbb{Z}_{\mathrm{T}} \circ \mathit{lift} \circ \mathit{lift}\\
&\qquad \mathit{tbind}\ m\ f \quad = \mathbb{Z}_{\mathrm{T}}\ \$\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ m \ggg\!= \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ f\\
&\qquad \mathit{tmixmap}\ f\ g = \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ (\mathit{tmixmap}\ f\ g)\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{tmixmap}\ g\ f) \circ \mathit{run}\mathbb{Z}_{\mathrm{T}}
\end{aligned}
$$

Focus The interesting behavior of $t_1 \triangleright t_2$, where it deviates from a plain monad transformer composition, lies in the methods of the monad classes: for looking up the method implementations it ignores (looks through) t_1 and only considers $t_2\ m$.

Consider the state monad $\mathbb{S}_{\mathrm{M}}$ defined by Figure 2. In the case of the monad zipper transformer $t_1 \triangleright t_2$, t_2 should be on focus and t_1 should be ignored. Thus adopting the uniform lifting functionality provided by the second instance would be the wrong thing to do. With such implementation, the definition of *stateM* would be equivalent to:

$$\mathit{stateM} = \mathit{isoMakeWith}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ \mathit{stateM}$$

using an auxiliary function

$$
\begin{aligned}
&\mathit{isoMakeWith}\ ::\ (\forall a.m\ a \to n\ a) \to (\forall a.n\ a \to m\ a)\\
&\qquad\qquad \to \mathit{MakeWith}\ s\ m \to \mathit{MakeWith}\ s\ n
\end{aligned}
$$

that changes a *MakeWith* dictionary using a given monad isomorphism. The above *stateM* code would merely adopt the $\mathbb{S}_{\mathrm{M}}$ implementation of $t_1\ (t_2\ m)$ for $(t_1 \triangleright t_2)\ m$ through the $(\mathbb{Z}_{\mathrm{T}}, \mathit{run}\mathbb{Z}_{\mathrm{T}})$ monad isomorhism, and thereby prefer the $\mathbb{S}_{\mathrm{M}}$ implementation of t_1 before any in $t_2\ m$. However, that is not the desired behavior for the monad zipper. Instead, using *liftMakeWith*, we lift the *stateM* implementation of $t_2\ m$ through t_1, *ignoring* any possible *stateM* implementation available for t_1.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{MonadT}\ t_1, \mathit{MonadT}\ t_2, \mathit{Monad}\ m,\\
&\quad \mathbb{S}_{\mathrm{M}}\ s\ (t_2\ m)) \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ ((t_1 \triangleright t_2)\ m)\ \textbf{where}\\
&\qquad \mathit{stateM} =\\
&\qquad\quad \mathit{isoMakeWith}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ (\mathit{liftMakeWith}\ \mathit{stateM})
\end{aligned}
$$

Example Now let's have a look at how to actually use the zipper monad. Consider the following two examples. The first example runs $put\ 1$ in the regular stack $\mathbb{S}_\mathrm{T}\ Int\ (\mathbb{S}_\mathrm{T}\ Int\ \mathbb{I})$, and hence updates the state of the topmost $\mathbb{S}_\mathrm{T}$ transformer. The second shifts the focus to the other $\mathbb{S}_\mathrm{T}$ transformer with a monad stack of the form $(\mathbb{S}_\mathrm{T}\ Int \rhd \mathbb{S}_\mathrm{T}\ Int)\ \mathbb{I}$.

$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ put\ 1$
$(((), 1), 0)$
$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{Z}_\mathrm{T}\ \$\ put\ 1$
$(((), 0), 1)$

On the surface, $run\mathbb{Z}_\mathrm{T}$ does not provide any expressive power over *lift*:

$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ lift\ \$\ put\ 1$
$(((), 0), 1)$

where $put\ 1$ has type $\mathbb{S}_\mathrm{T}\ Int\ \mathbb{I}\ ()$. Note that the topmost $\mathbb{S}_\mathrm{T}\ Int$ does not appear in this type. In contrast, unlike *lift*, $run\mathbb{Z}_\mathrm{T}$ does not lose information about any monadic layers. Despite the deceiving similarity between $run\mathbb{Z}_\mathrm{T}$ and *lift*, we will see that $run\mathbb{Z}_\mathrm{T}$ has great advantages when it comes to modular components. First though, we further develop the correspondence between Huet's zipper and our monad zipper.

Relative Navigation Suppose we have a monad transformer stack $t_1\ (t_2 ... (t_n\ m))$. Then the focus lies by default on the topmost transformer t_1. Analogous to what the *zipper* function does with *Stack*, we can change this monad transformer stack into explicit zipper form:

$$zipper :: t\ m\ a \to (\mathbb{I}_\mathrm{T} \rhd t)\ m\ a$$
$$zipper = \mathbb{Z}_\mathrm{T} \circ \mathbb{I}_\mathrm{T}$$

where the identity monad transformer $\mathbb{I}_\mathrm{T}$ acts as the *Top* sentinel. However, this change is entirely unnecessary: $t\ m\ a$ and $(\mathbb{I}_\mathrm{T} \rhd t)\ m\ a$ have exactly the same automatic lifting behavior. Indeed, we have added $\mathbb{I}_\mathrm{T}$ to subsequently ignore it again with $\mathbb{I}_\mathrm{T} \rhd t$.

The monad zipper becomes useful only when we shift the focus away from t_1 to t_2. We have already seen that $\mathbb{Z}_\mathrm{T}$ accomplishes that shift of focus, but how can we navigate further down, and back up?

Let us start with moving the focus one step further down:

$$step2to3 :: (t_1 \rhd t_2)\ (t_3\ m)\ a \to (? \rhd t_3)\ m\ a$$

What should come in the place of the question mark? Following Huet's zipper, we should push t_2 on a reversed stack that already contains t_1. Pleasingly, if we denote this reversed stack as $t_1 \rhd t_2$, we obtain the following very simple implementation for *step2to3*:

$$step2to3 :: (t_1 \rhd t_2)\ (t_3\ m)\ a \to (t_1 \rhd t_2 \rhd t_3)\ m\ a$$
$$step2to3 = \mathbb{Z}_\mathrm{T}$$

A further step down:

$$step3to4 :: (t_1 \rhd t_2 \rhd t_3)\ (t_4\ m)\ a \to (t_1 \rhd t_2 \rhd t_3 \rhd t_4)\ m\ a$$
$$step3to4 = \mathbb{Z}_{\mathrm{T}}$$

The pattern should now be obvious. A single step down at any position in the stack is defined as:

$$\downarrow :: t_1\ (t_2\ m)\ a \to (t_1 \rhd t_2)\ m\ a$$
$$\downarrow = \mathbb{Z}_{\mathrm{T}}$$

Stepping back up is similar:

$$\uparrow :: (t_1 \rhd t_2)\ m\ a \to t_1\ (t_2\ m)\ a$$
$$\uparrow = run\mathbb{Z}_{\mathrm{T}}$$

such that $\downarrow \circ \uparrow \equiv id$ and $\uparrow \circ \downarrow \equiv id$ hold.

3.3 Abstraction with the Zipper

How does the monad zipper solve the monad stack abstraction problem, and avoid clashing monad transformer instances? Simple, we shift the focus on a different layer in the stack for each feature. That way the different monad transformer instances do not all have to be at the top of the stack.

A simple application of this idea consists of defining a combinator $\otimes$ that shifts each monadic layer one level to the right.

$$\begin{aligned}(\otimes)\ ::\ & Mixin\ (a \to t_1\ (t_2\ m)\ b) \\ & \to Mixin\ (a \to (t_1 \rhd t_2)\ m\ b) \\ & \to Mixin\ (a \to t_1\ (t_2\ m)\ b) \\ c1 \otimes c2 = {} & \lambda proceed\ x \to \\ & c1\ (\uparrow \circ c2\ (\downarrow \circ proceed))\ x\end{aligned}$$

Here component *c1* focusses on the current layer, and *c2* looks one position down – that's why we have to bring *proceed* down ($\downarrow$) to its level and shift the whole back up ($\uparrow$) to the current level.

This combinator is very useful whenever we have a set of features that uses a disjoint set of monads (that is, each feature will use different monads). No additional work is needed to make the two state transformers of *prof* and *memo* happily coexist.

$$profmemofib :: Int \to \mathbb{S}_{\mathrm{T}}\ Int\ (\mathbb{S}_{\mathrm{T}}\ (Map\ Int\ Int)\ (\mathbb{I}_{\mathrm{T}}\ \mathbb{I}))\ Int$$
$$profmemofib = fix\ (prof \otimes memo \otimes fib2')$$

Note that every component has its own state transformer, notably $\mathbb{I}_{\mathrm{T}}$ for *fib2′*, and we use the base monad $\mathbb{I}$ at the bottom of the stack.

3.4 Effect Encapsulation

To evaluate *profmemofib*, the use has to supply the appropriate run functions:

$$n = run\mathbb{I} \circ run\mathbb{I}_{\mathrm{T}} \circ eval\mathbb{S}_{\mathrm{T}}\ empty \circ run\mathbb{S}_{\mathrm{T}}\ 0\ \$\ profmemofib\ 7$$

We can encapsulate the components more tightly by bundling them with their run functions and hiding the effect types.

$$\begin{aligned}
&\textbf{data}\ Component\ f\ a\ b = \forall\ t_2.MonadT\ t_2 \Rightarrow \\
&\quad Component\ \{\, behavior\ :: \forall\ t_1\ m.(MonadT\ t_1, Monad\ m) \\
&\qquad\qquad\qquad \Rightarrow Mixin\ (a \to (t_1 \rhd t_2)\ m\ b) \\
&\qquad\quad , run\qquad :: \forall\ m\ x.Monad\ m \Rightarrow t_2\ m\ x \to m\ (Run\ f\ x)\,\} \\
&\textbf{type family}\ Run\ f\ x
\end{aligned}$$

In the *Component* type definition, t_2 is an existentially quantified type. That means it is hidden from users of the component. In contrast, m and t_1 are universally quantified. This means that the user of the component gets to choose, and the component must work for all possible choices. In other words, every component is only aware of its own effects.

The *behavior* field contains the mixin, while *run* captures the effect evaluation. The latter should handle any return type x, and is allowed to change that return type according to a type family *Run*. The need for this becomes clear when we look at particular component implementations:

$$\begin{aligned}
&fibC :: Component\ ()\ Int\ Int \\
&fibC = Component\ \{\, behavior = fib2', run = run\mathbb{I}_{\mathrm{T}}\,\} \\
&memoC :: Component\ ()\ Int\ Int \\
&memoC = Component\ \{\, behavior = memo, run = eval\mathbb{S}_{\mathrm{T}}\ empty\,\}
\end{aligned}$$

Both *fibC* and *memoC* leave the result type unmodified. This is captured in the type family instance.

$$\textbf{type instance}\ Run\ ()\ x = x$$

In contrast, the profiling component augments the result type with the profiling information.

$$\begin{aligned}
&profC :: Component\ (RPair\ Int)\ Int\ Int \\
&profC = Component\ \{\, behavior = prof, run = run\mathbb{S}_{\mathrm{T}}\ 0\,\} \\
&\textbf{data}\ RPair\ s \\
&\textbf{type instance}\ Run\ (RPair\ s)\ x = (x, s)
\end{aligned}$$

The operators $\times$ and *fixC* are the component counterparts of the mixin operators $\otimes$ and *fix*.

$$\begin{aligned}
&fixC :: Component\ f\ a\ b \to (a \to Run\ f\ b) \\
&fixC\ (Component\ bhC\ runC) = run\mathbb{I} \circ runC \circ run\mathbb{I}_{\mathrm{T}} \circ run\mathbb{Z}_{\mathrm{T}} \circ fix\ bhC
\end{aligned}$$

Note that *fixC* instantiates the top and bottom of them monad stack with identity effects $\mathbb{I}_{\mathrm{T}}$ and $\mathbb{I}$ respectively.

$$
\begin{aligned}
&(\times) :: \mathit{Component}\ f_1\ a\ b \to \mathit{Component}\ f_2\ a\ b \to \mathit{Component}\ (f_1, f_2)\ a\ b \\
&(\mathit{Component}\ \mathit{bh1}\ \mathit{run1}) \times (\mathit{Component}\ \mathit{bh2}\ \mathit{run2}) = \mathit{Component}\ \{ \\
&\quad \mathit{behavior} = \lambda \mathit{proceed} \to \\
&\qquad \mathbf{let}\ \mathit{proceed}' = \mathbb{Z}_{\mathrm{T}} \circ \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ \mathbb{Z}_{\mathrm{T}} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{proceed} \\
&\qquad\quad \mathit{bh2}' \quad = \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{bh2}\ \mathit{proceed}' \\
&\qquad\quad \mathit{bh1}' \quad = \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{bh1}\ \mathit{bh2}' \\
&\qquad \mathbf{in}\ \mathit{bh1}' \\
&\quad , \mathit{run} \quad = \mathit{run2} \circ \mathit{run1} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \} \\
&\mathbf{type\ instance}\ \mathit{Run}\ (f_1, f_2)\ x = \mathit{Run}\ f_2\ (\mathit{Run}\ f_1\ x)
\end{aligned}
$$

The implementation of component composition $\times$ is particularly complicated by attaining the appropriately focused shapes of the same monad stack for the different components and their resulting composition.

Finally, we may compose a number of different functions as follows:

$$
\begin{aligned}
&\mathit{proffib}' :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{proffib}' = \mathit{fixC}\ (\mathit{profC} \times \mathit{fibC}) \\
&\mathit{profmemofib}' :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{profmemofib}' = \mathit{fixC}\ (\mathit{profC} \times \mathit{memoC} \times \mathit{fibC})
\end{aligned}
$$

$$
\begin{aligned}
&> \mathit{proffib}'\ 20 \\
&(6765, 21891) \\
&> \mathit{profmemofib}'\ 20 \\
&(6765, 39)
\end{aligned}
$$

4 Discussion

After this exercise on abstraction, it is time to reflect on some of the design choices and summarize the main ideas.

4.1 Monad Transformers and Data Types à la Carte

As shown in Section 2 composing modular components with effects is not straightforward. The approach described in this paper can be viewed as an improvement of the existing techniques.

The approach of (4) to modular interpreters is an important step towards the goal of modularizing interpreters (and programs in general). However their approach does not support separate compilation nor the concurrent development of several interpreters, because all the features are entangled through hard references.

The Data types à la Carte approach (5) avoids these hard references: it shows how to abstract away from the concrete compositions of datatypes. Unlike Liang et al., Swierstra does not consider the issue of modular implementations of *effectful* features of an interpreter. He does, however, apply his technique to free monads, obtaining a modular way to combine different monads. This provides an alternative to monad transformers, but we expect similar issues to the ones identified in Section 2 to occur for stacks of free monads. Thus, a monad zipper suitably adapted to stacks of free monads would be desirable.

Unlike Liang et al. and Swierstra, we use open recursion instead of type classes. Type classes are very good for the ultimate automation as we do not even have to bother explicitly composing features. However, this approach does not allow multiple implementations for the same feature to coexist, since type classes do not permit more than one instance per (feature) type. At a relatively small cost of explicitly composing features by hand we gain increased flexibility, expressive power and ability to reuse components.

4.2 Other Stacks for The Zipper

Although the Monatron library is used in this paper to present the monad zipper, it is certainly possible to use other monad transformer libraries such as the MTL, which is a library inspired by the original design proposed by (4). We have two main reasons to prefer Monatron over the MTL.

- The first reason is that in the MTL, the class for monad transfomers
  ```
  class MonadTrans t where
    lift :: m a → t m a
  ```
  provides only the *lift* method. However, in order to lift the operations of the various monads through the monad zipper an operation like the *tmixmap* provide by Monatron is necessary. MTL is not fundamentally incompatible; one work-around consists of adding *tmixmap* in a new subclass of *MonadTrans*.
- The second, more fundamental, reason to prefer Monatron over the MTL is that the MTL design prevents certain operations from being lifted. In particular, as noted by (6), the *listen* operation of the writer monad seems to be impossible to lift. This would preclude the use of writer monads, which is not desirable. Nevertheless, if we would be willing to give up the *listen* operation or the writer monad, then it would be possible to use the monad zipper in the MTL.

While we focus here on the range of monad transformers available in Monatron, it should be possible to use the monad zipper in stacks of free monads. It would be also be interesting to adapt the zipper to other effect stacks such as *applicative functors* (7) or *arrows* (8).

4.3 Open Recursion

Cook (2) was one of the first to propose open recursion as a means to extend existing behavior in an OOP inheritance setting. Recently, Oliveira et al. (9)

have done the same to model effectful aspect oriented programming advice using monads to model effects.

The modularity concerns addressed in these works are orthogonal to our own: they are concerned with augmenting the behavior of a feature, rather than adding more features. In fact, they can be readily combined with our work to, for example, add profiling as a separate component that can be (re-)used to advise different features.

5 Conclusion

With this paper we have shown that highly modular *and* effectful systems can be realized in Haskell. Our solution borrows heavily from the literature, in particular Liang's modular interpreters and Swierstra's data types à la carte, but the monad zipper is the key ingredient that ties everything together. The code is available on Hackage, as part of the Monatron library.

In ongoing work we are investigating two applications of our approach:

1. solutions to the expression problem such as Data Types à la Carte (5), and
2. combinators for compositional search heuristics as outlined in the Monadic Constraint Programming framework (10).

At a more abstract level, we would like to build a name-based approach on top of our structural technique, comparable to named variables versus De Bruijn indices.

Acknowledgements

We are grateful to Jeremy Gibbons, heisenbug, Mauro Jaskelioff, Wonchan Lee, Wouter Swierstra, Phil Wadler and Stephanie Weirich for their help and feedback on an earlier version of this text.

Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders. Bruno Oliveira is supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant number R11-2008-007-01002-0.

Bibliography

[1] Jaskelioff, M.: Monatron: An extensible monad transformer library. In: IFL '08: Symposium on Implementation and Application of Functional Languages. (2008)

[2] Cook, W.R.: A Denotational Semantics of Inheritance. PhD thesis, Brown University (1989)

[3] Huet, G.: Functional Pearl: The Zipper. Journal of Functional Programming **7**(5) (September 1997) 549–554

[4] Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL'95. (1995)

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

[6] Jaskelioff, M.: Modular monad transformers. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2009) 64–79

[7] Mcbride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1) (2008) 1–13

[8] Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37** (1998) 67–111

[9] Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: Effectiveadvice: Disciplined advice with explicit effects. In: AOSD '10: ACM SIG Proceedings of the 9th International Conference on Aspect-Oriented Software Development. (2010)

[10] Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. J. Funct. Program. **19**(6) (2009) 663–697