Functional Pearl: The Monad Zipper

Tom Schrijvers
Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A
3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be

Bruno C. d. S. Oliveira
ROSAEC Center, Seoul National University
Room 215, Building 138
599 Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea
bruno@ropas.snu.ac.kr

Abstract

Limitations of monad stacks get in the way of developing highly modular programs with effects. This pearl 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Haskell; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks, patterns, recursion

General Terms Design, Languages

Keywords Monads, monad transformers, zipper, modularity, mixins, expression problem

1. Introduction

As Functional Programmers it would seem that problem solving invariably involves writing an interpreter of one kind or another. Adapting an existing interpreter is of course better than writing a new one from scratch. That is why Liang et al. (1995)'s technique for *modular interpreters* is so appealing. A key idea of the modular interpreters technique, is that each of the features of an interpreter is written as a separate functor. For example, a simple interpreter featuring variables and addition has the following two functors:

```
data Var e = Var String
data Add e = Add e e
```

which are combined using the disjoint sum

```
data (⊕) f g e = Inl (f e) | Inr (g e)
```

The fixpoint assembles a complete expression type from the desired combination $Var \oplus Lit$:

```
newtype Fix f = In{ out :: f (Fix f)}
type Expr0    = Fix (Add ⊕ Var)
```

Each of the features provides a separate evaluation function

```
evalVar :: Var Expr0 → M Int
evalVar (Var v) =
  do env ← get
     case lookup v env of
       Nothing → throw "Variable does not exist!"
       Just x  → return x
evalAdd :: Add Expr0 → M Int
evalAdd (Add l r) =
  do x ← eval0 l
     y ← eval0 r
     return (x + y)
```

The main evaluation function dispatches to the appropriate feature's evaluation function

```
eval0 :: Expr0 → M Int
eval0 (In e) = case e of
  Inl e → evalAdd e
  Inr e → evalVar e
```

The M monad fills in the requirements for the side-effects of features. In this case a state monad carries the variable environment around and an exception monad reports uses of undefined variables. *Monad transformers* (Liang et al. 1995) combine these separate monads into a stack-like monadic structure.

```
type Env = [(String, Int)]
type M   = S_T Env (E_T String I)
```

The monad transformers $\mathbb{S}_T$ and $\mathbb{E}_T$ represent, respectively, the state and exception monad transformers; and $\mathbb{I}$ is the identity monad.

For convenience smart constructors can be used to construct values of $Expr_0$

```
add0 :: Expr0 → Expr0 → Expr0
add0 e1 e2 = In (Inl (Add e1 e2))
var0 :: String → Expr0
var0 v     = In (Inr (Var v))
```

making the interpreter usable in essentially the same way as a conventional (non-modular) interpreter

```
client = eval0 (var0 "x" `add0` var0 "y")
```

What is great about this modular approach is that the code of the existing features does not need to be touched, if we want to add, say, integer literals. All that is needed is the code for the new feature

```
data Lit e = Lit Int
evalLit :: Lit Expr1 → M Int
evalLit (Lit l) = return l
```

and some updates in the composition and smart constructors code

ICFP '10 September 27-29, Baltimore.

```
type Expr₁ = Fix (Add ⊕ Var ⊕ Lit)

eval₁ :: Expr₁ → Int
eval₁ (In e) = case e of
  Inl e         → evalAdd e
  Inr (Inl e)   → evalVar e
  Inr (Inr e)   → evalLit e

add₁ e₁ e₂ = In (Inl (Add e₁ e₂))
var₁ v     = In (Inr (Inl (Var v)))
lit₁ l     = In (Inr (Inr (Lit l)))
```

The catch Unfortunately, the modularity is not as good as it seems. Even though we do not have to touch the *evalVar* and *evalAdd* code for this addition, we do have to recompile it. The reason is that the original definition expressions $Expr_0$, on which the features depend, has been updated into $Expr_1$. The monad type M forms a similar dependency that forces recompilation when updated.

Another important drawback is that the features cannot be used in two or more languages at the same time. Again the two dependencies are in our way: the same expression and monad types are used across all components. Thus two features that use different expression and monad types cannot reuse the same evaluation code. Another dependency is present in *evalAdd*: there is only one *eval* function for recursive calls.

Essentially the whole system is entangled with hard, mutual dependencies, making separate compilation and reuse across multiple different interpreters difficult.

Can we get rid of these hard dependencies and make our interpreter more modular?

2. Dependency Abstraction

The solution to hard-wired dependencies is of course abstraction, which comes in different flavors in Functional Programming.

2.1 Function Abstraction

We relax the *eval* dependency by parametrization, making *eval* an argument of *evalAdd*:

```
evalAdd eval (Add l r) =
  do x ← eval l
     y ← eval r
     return (x + y)
```

This yields the type signature

```
evalAdd :: (Expr → M Int) → (Add Expr → M Int)
```

Other evaluation functions are written similarly for uniformity, even though they do not involve recursive calls

```
evalVar :: (Expr → M Int) → (Var Expr → M Int)
evalLit :: (Expr → M Int) → (Lit Expr → M Int)
```

Going one abstraction step further, the more general type of this *open recursion* (Cook 1989) pattern is captured by the following type synonym:

```
type Open e f r = (e → r) → (f e → r)

evalAdd :: Open Expr Add (M Int)
evalVar :: Open Expr Var (M Int)
evalLit :: Open Expr Lit (M Int)
```

Rather than a big multiway **case** expression for *eval* the features can be more conveniently composed using the ⊛ combinator

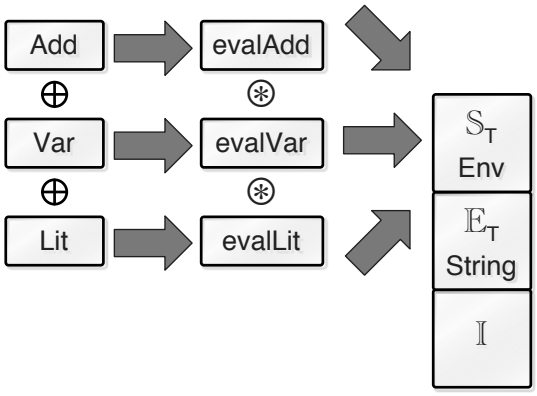

Figure 1. The structural composition of the evaluator

```
class (f ≼ g) where
  inj :: f a → g a
instance Functor f ⇒ f ≼ f where
  inj = id
instance (Functor g, Functor f)
         ⇒ f ≼ (f ⊕ g) where
  inj = Inl
instance (Functor g, Functor h, Functor f, f ≼ g)
         ⇒ f ≼ (h ⊕ g) where
  inj = Inr ∘ inj
inject :: (f ≼ g) ⇒ f (Fix g) → Fix g
inject = In ∘ inj
```

Figure 2. Injections for the smart constructors.

```
(⊛) :: Open e f r → Open e g r → Open e (f ⊕ g) r
eval_f ⊛ eval_g = λeval e →
  case e of
    Inl el → eval_f eval el
    Inr er → eval_g eval er
```

and the open recursion can be closed with a fixpoint combinator

```
fix :: Open (Fix f) f r → (Fix f → r)
fix f = let this = f this ∘ out
        in this
```

yielding a compact *eval* function

```
eval = fix (evalAdd ⊛ evalVar ⊛ evalLit)
```

What makes this composition so convenient is that the structure of the evaluation function follows exactly the structure of expression type

```
type Expr₁ = Fix (Add ⊕ Var ⊕ Lit)
```

Figure 1 illustrates the connections between the feature types, feature implementations and monad stack.

2.2 Constructor Abstraction

It is possible to make the smart constructors smarter by using a technique described in the "*Data Types à la Carte*" (Swierstra 2008) approach. At the essence of these smarter constructors is an overloaded injection function *inject*, which automatically lifts a constructor to the right layer in a disjoint sum of functors. The

relevant code is shown in Figure 2. For each datatype representing a feature one functor instance is needed:

```
instance Functor Lit where
  fmap _ (Lit l)       = Lit l
instance Functor Var where
  fmap _ (Var v)       = Var v
instance Functor Add where
  fmap f (Add e1 e2) = Add (f e1) (f e2)
```

and we also need one instance for functor composition:

```
instance (Functor f, Functor g) ⇒
  Functor (f ⊕ g) where
    fmap f (Inl x) = Inl (fmap f x)
    fmap f (Inr y) = Inr (fmap f y)
```

Using *inject* smarter constructors can be defined for the various different types of expressions as follows:

```
lit :: (Lit ≼ g)    ⇒ Int → Fix g
lit l               = inject (Lit l)

var :: (Var ≼ g)  ⇒ String → Fix g
var v             = inject (Var v)

add :: (Add ≼ g) ⇒ Fix g → Fix g → Fix g
add e1 e2 = inject (Add e1 e2)
```

The advantage of these constructors versus the ones defined in Section 1, is that they do not depend on a particular type of expressions. Instead they can construct values of multiple different expression types such as $Expr_0$ or $Expr_1$.

2.3 Type Abstraction

We can easily further abstract the types of the features' evaluation functions by generalizing the concrete types *Expr* and M to type variables e and m. In fact, modulo the type synonym, type inference can do this for us automatically.

```
evalLit  :: Monad m
         ⇒ Open e Lit (m Int)
evalVar  :: (S_M Env m, E_M String m)
         ⇒ Open e Var (m Int)
evalAdd  :: Monad m
         ⇒ Open e Add (m Int)
```

Note that the type class constraints capture the requirements on the monad m: $\mathbb{S}_{\text{M}}$ *Env* m means that m should provide a state of type *Env*, and $\mathbb{E}_{\text{M}}$ *String* m means that m should support exceptions of type *String*.

Figure 3 shows a quick reference guide for the various monad transformers, classes and operations used throughout this text. All of these come from the ***Monatron library*** (Jaskelioff 2008), which is the monad transformer library used in this paper.

2.4 Client Code

Now we can compose different expression languages with their separate evaluation functions side-by-side:

```
type Expr0 = Fix (Add ⊕ Var)
type Expr1 = Fix (Add ⊕ Var ⊕ Lit)
type Expr2 = Fix (Add ⊕ Lit)

eval0 :: Expr0 → M Int
eval0 = fix (evalAdd ⊛ evalVar)

eval1 :: Expr1 → M Int
eval1 = fix (evalAdd ⊛ evalVar ⊛ evalLit)

eval2 :: Expr2 → I Int
eval2 = fix (evalAdd ⊛ evalLit)
```

In order to conveniently run the evaluation functions, the monadic layers are removed:

```
run0 :: Env → Expr0 → Int
run0 e = unwrap e ∘ eval0

run1 :: Env → Expr1 → Int
run1 e = unwrap e ∘ eval1

run2 :: Expr2 → Int
run2   = runI ∘ eval2
```

Here, *unwrap* is defined as follows:

```
unwrap e = handler ∘ runI ∘ runE_T ∘ runS_T e

handler (Left x)  = error x
handler (Right t) = fst t
```

Constructor abstraction comes into play when some expressions are defined

```
expr12 :: (Lit ≼ g, Add ≼ g) ⇒ Fix g
expr12 = add (lit 3) (lit 4)

expr1 :: (Lit ≼ g, Add ≼ g, Var ≼ g) ⇒ Fix g
expr1  = add expr12 (var "x")
```

The expression $expr_{12}$ requires a functor supporting *Lit* and *Add*. Since both $Expr_1$ and $Expr_2$ satisfy these requirements, the following programs are valid:

```
env = [("x", 6)]

p1 = run1 env expr12
p2 = run2 expr12
```

In contrast, $expr_1$ requires functors that are only supported by by $Expr_1$. Consequently only run_1 can be used to evaluate $Expr_1$:

```
p3 = run1 env expr1
```

2.5 Mission Accomplished?

That worked wonderfully well: standard abstraction mechanisms give us separate compilation and reusability of interpreter features. So, mission accomplished? Actually, on closer inspection, the monad abstraction does not work well at all.

Let's add a memory feature that allows storing the result of an expression in a register and retrieving it again.

```
data Mem e = Store e | Retrieve
type Reg   = Int
evalMem :: S_M Reg m ⇒ Open e Mem (m Int)
evalMem eval (Store e) =
  do n ← eval e
     put n
     return n
evalMem eval Retrieve  = get
```

The type checker complains when we add this feature to our existing expression language as follows

```
type Expr3 = Fix (Mem ⊕ Var ⊕ Lit)
type M3    = S_T Reg (S_T Env (E_T String I))

eval3 :: Expr3 → M3 Int
eval3 = fix (evalMem ⊛ evalVar ⊛ evalLit)
```

The type checker complaint is that *Reg* and *Env* 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 $\mathbb{S}_{\text{T}}$, which

```
  -- identity monad
newtype 𝕀 a
𝕀       :: a → 𝕀 a
run𝕀    :: 𝕀 a → a
  -- identity monad transformer
newtype 𝕀T m a
𝕀T      :: m a → 𝕀T m a
run𝕀T   :: 𝕀T m a → m a
```

```
  -- reader monad transformer
newtype ℝT e m a
ℝT      :: (e → m a) → ℝT e m a
runℝT   :: e → ℝT e m a → m a
  -- state monad transformer
newtype 𝕊T s m a
𝕊T      :: (s → m (a, s)) → 𝕊T s m a
run𝕊T   :: s → 𝕊T s m a → m (a, s)
  -- exception monad transformer
newtype 𝔼T x m a
𝔼T      :: m (Either x a) → 𝔼T x m a
run𝔼T   :: 𝔼T x m a → m (Either x a)
```

```
  -- reader monad class
class Monad m ⇒ ℝM e m | m → e
ask     :: ℝM e m ⇒ m e
  -- state monad class
class Monad m ⇒ 𝕊M s m | m → s
get     :: 𝕊M s m ⇒ m s
put     :: 𝕊M s m ⇒ s → m ()
  -- exception monad class
class Monad m ⇒ 𝔼M x m | m → x
throw   :: 𝔼M x m ⇒ x → m a
```

Figure 3. Monatron quick reference.

happens to contain a *Reg* value. This is the right thing to do for *evalMem*, but wrong for *evalVar* that expects a value of type *Env*.

This type error is only a symptom of the real problem though. Namely, we expect the *get* calls in *evalMem* and *evalVar* 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 $\mathbb{S}_T$ 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 potentialy 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 polute the code and they interact well with abstraction, implicitly lifting the monad into the right layer. Unfortunatelly, 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 $\mathbb{I}$.

Huet (1997) 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:

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

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 Haskell newtype indicates that no actual structural change to the monad stack $t_1\ (t_2\ m)$ takes place.

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:

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

In Monatron, a monad transformer 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:

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

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.

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 $t_2\ m$.

For instance, consider the state monad with its methods for reading and writing the state. Monatron provides an explicit *dictionary* type

$$\textbf{type}\ MakeWith\ s\ m$$

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:

$$\begin{aligned}
&getX :: Monad\ m \Rightarrow MakeWith\ s\ m \to m\ s\\
&putX :: Monad\ m \Rightarrow MakeWith\ s\ m \to s \to m\ ()
\end{aligned}$$

A dictionary can be generated for $\mathbb{S}_\mathrm{T}\ s\ m$

$$makeWithStateT :: Monad\ m \Rightarrow MakeWith\ s\ (\mathbb{S}_\mathrm{T}\ s\ m)$$

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:

$$\begin{aligned}
liftMakeWith :: &(Monad\ m, MonadT\ t) \Rightarrow\\
&MakeWith\ z\ m \to MakeWith\ z\ (t\ m)
\end{aligned}$$

For convenience Monatron uses Haskell's type class mechanism to make the dictionaries implicit. Figure 4 shows how this is done for the state monad. The first instance implements the specific functionality for the $\mathbb{S}_\mathrm{T}$ 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

$$\begin{aligned}
&\textbf{class}\ Monad\ m \Rightarrow \mathbb{S}_\mathrm{M}\ z\ m \mid m \to z\ \textbf{where}\\
&\quad stateM :: MakeWith\ z\ m\\
&\textbf{instance}\ Monad\ m \Rightarrow \mathbb{S}_\mathrm{M}\ z\ (\mathbb{S}_\mathrm{T}\ z\ m)\ \textbf{where}\\
&\quad stateM = makeWithStateT\\
&\textbf{instance}\ (\mathbb{S}_\mathrm{M}\ z\ m, MonadT\ t) \Rightarrow \mathbb{S}_\mathrm{M}\ z\ (t\ m)\ \textbf{where}\\
&\quad stateM = liftMakeWith\ stateM\\
&get :: \mathbb{S}_\mathrm{M}\ z\ m \Rightarrow m\ z\\
&get = getX\ stateM\\
&put :: \mathbb{S}_\mathrm{M}\ z\ m \Rightarrow z \to m\ ()\\
&put = putX\ stateM
\end{aligned}$$

Figure 4. Overloading of state operations in Monatron

in essentially the same way: one instance provides the functionality specific to the particular monad in question; whereas another instance provides uniform lifting.

The key idea 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 lifing functionality provided by the second instance would be the wrong thing to do. With such implementation, the definition of $stateM$ would be equivalent to:

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

using an auxiliary function

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

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 $(\mathbb{Z}_\mathrm{T}, run\mathbb{Z}_\mathrm{T})$ 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.

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

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 $(\mathbb{S}_\mathrm{T}\ Int \triangleright \mathbb{S}_\mathrm{T}\ Int)\ \mathbb{I}$.

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

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

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

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:

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

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*:

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

A further step down:

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

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

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

Stepping back up is similar:

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

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.

$$\begin{aligned}&(\otimes)\ ::\ Open\ e\ f\ (t_1\ (t_2\ m)\ a)\\ &\quad \to Open\ e\ g\ ((t_1 \rhd t_2)\ m\ a)\\ &\quad \to Open\ e\ (f \oplus g)\ (t_1\ (t_2\ m)\ a)\\ &eval_f \otimes eval_g = \lambda eval\ e \to\\ &\quad \mathbf{case}\ e\ \mathbf{of}\\ &\qquad Inl\ el \to eval_f\ eval\ el\\ &\qquad Inr\ er \to \uparrow (eval_g\ (\downarrow \circ eval)\ er)\end{aligned}$$

Here the $eval_f$ feature focusses on the current layer, and $eval_g$ looks one position down – that's why we have to bring *eval* 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 *evalMem* and *evalVar* happily coexist.

$$\begin{aligned}&eval_3 :: Expr_3 \to M_3\ Int\\ &eval_3 = fix\ (evalMem \otimes evalVar \otimes evalLit)\end{aligned}$$

Note that zipper again nicely preserves the structural correspondence between the expression type and the definition of the evaluation function. More pleasingly, the zipper extends this structural correspondence to include the monad stack as well (see Figure 5), as opposed to the previous approach where the same (view of the) monad stack is shared by all features (see Figure 1).

Actually, in the above composition the $\mathbb{E}_\mathrm{T}\ String$ transformer required by *Var* is lined up with (but not used by) *Lit*. In this configuration, that is not a problem, but it does become a problem when reversing the order of features

$$\mathbf{type}\ Expr_{3b} = Fix\ (Lit \oplus Var \oplus Mem)$$

It is a cleaner solution to group the two transformers used by *Var* into a single one, using the composition operator $\bullet$ defined in Figure 6.[1]

$$\begin{aligned}&\mathbf{type}\ M_{3b} = \mathbb{I}_\mathrm{T}\ ((\mathbb{S}_\mathrm{T}\ Env \bullet \mathbb{E}_\mathrm{T}\ String)\ (\mathbb{S}_\mathrm{T}\ Reg\ \mathbb{I}))\\ &eval_{3b} :: Expr_{3b} \to M_{3b}\ Int\\ &eval_{3b} = fix\ (evalLit \otimes evalVar \otimes evalMem)\end{aligned}$$

Figure 7 depicts the new structure: every feature type is neatly aligned with one evaluation function and one (possibly composite or identity) transformer.

[1] Contrast the implementation of $(\bullet)$ with that of $(\rhd)$.

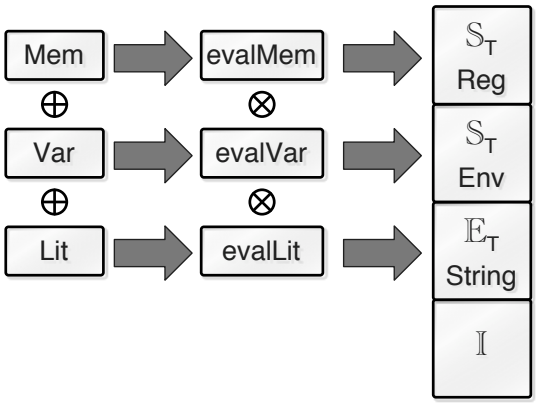

Figure 5. Structural composition with the monad zipper.

$$\begin{aligned}&\mathbf{newtype}\ (t_1 \bullet t_2)\ m\ a = \mathbb{C}_\mathrm{T}\{run\mathbb{C}_\mathrm{T} :: t_1\ (t_2\ m)\ a\}\\ &\\ &\mathbf{instance}\ (MonadT\ t_1, MonadT\ t_2)\\ &\quad \Rightarrow MonadT\ (t_1 \bullet t_2)\ \mathbf{where}\\ &\qquad lift\ \ \ \ \ \ \ \ = \mathbb{C}_\mathrm{T} \circ lift \circ lift\\ &\qquad tbind\ m\ f\ \ \ = \mathbb{C}_\mathrm{T}\ \$\ run\mathbb{C}_\mathrm{T}\ m \gg\!= run\mathbb{C}_\mathrm{T} \circ f\\ &\qquad tmixmap\ f\ g = \mathbb{C}_\mathrm{T} \circ tmixmap\ (tmixmap\ f\ g)\\ &\qquad\qquad\qquad\qquad\qquad\qquad (tmixmap\ g\ f) \circ run\mathbb{C}_\mathrm{T}\\ &\mathbf{instance}\ (Monad\ m, MonadT\ t_1, MonadT\ t_2,\\ &\quad \mathbb{S}_\mathrm{M}\ s\ (t_1\ (t_2\ m))) \Rightarrow \mathbb{S}_\mathrm{M}\ s\ ((t_1 \bullet t_2)\ m)\ \mathbf{where}\\ &\qquad stateM = isoMakeWith\ \mathbb{C}_\mathrm{T}\ run\mathbb{C}_\mathrm{T}\ stateM\end{aligned}$$

Figure 6. Definition of monad transformer composition.

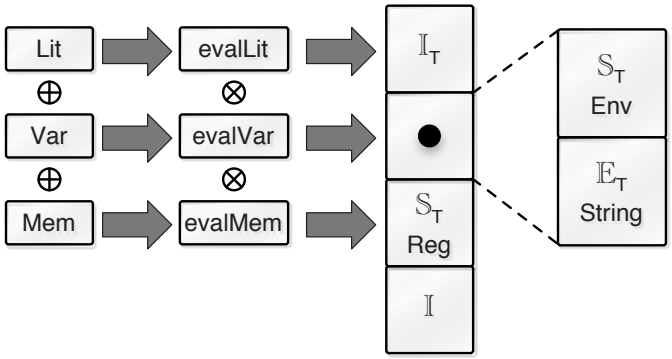

Figure 7. Structural composition with the monad zipper and transformer grouping

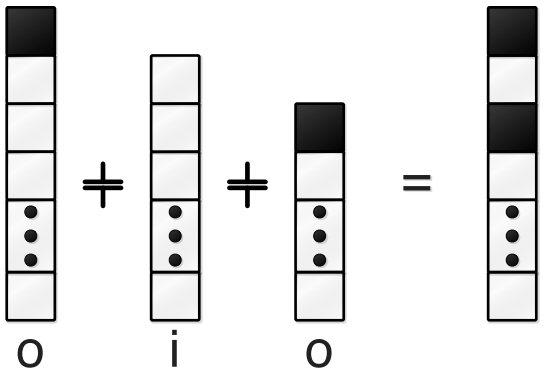

Figure 8. The $o \dotplus i \dotplus o$ mask

4. A Mask with Zippers

We have used the monad zipper to ignore a prefix of the monad stack. What if we want to ignore other, non-prefix, parts of the monad stack? The monad zipper can do that too, if we use it in the middle of the monad stack, rather than only at the top. Using the *Stack* - *Zipper* analogy, this would amount to merging the two datatypes into one:

$$\begin{aligned}\mathbf{data}\ ZStack = &\ Push \quad\quad\quad\quad Trans\ ZStack\\ \mid &\ Zipper\ Path\ Trans\ ZStack\\ \mid &\ Bottom\ Monad\end{aligned}$$

The *Zipper* constructor has almost the same role as the the *Push* constructor: to add another transformer *Trans* on top of a stack. The main difference is that is also carries a reversed path of transformers that are not part of the spine of the stack – to be ignored. For instance, $Push\ T_1\ (Push\ T_2\ (Push\ T_3\ M_1))$ is a regular stack with three transformers, while T_2 is ignored in $Push\ T_1\ (Zipper\ (Top\ T_2)\ T_3\ M_1)$.

With the actual monad zipper and transformers we do not need to repeat this merging process. They work together out of the box: $t_1\ (t_2\ (t_3\ m_1))$ is a regular monad stack and $t_1\ ((t_2 \triangleright t_3)\ m_1)$ ignores t_2.

4.1 Monad Masking Language

To simplify masking, we define a little language $(m \bowtie n)$, where n is the *masked view* of m, with primitive masks

$$\begin{aligned}&i :: Monad\ m \Rightarrow m \bowtie m\\ &o :: (Monad\ m, MonadT\ t_1, MonadT\ t_2)\\ &\quad \Rightarrow t_1\ (t_2\ m) \bowtie (t_1 \triangleright t_2)\ m\end{aligned}$$

where i means as much as "I want to see the current layer of the monad stack" and o means "I don't want to see the current layer".

The $(\dotplus)$ combinator provides vertical composition

$$\begin{aligned}&(\dotplus) :: (Monad\ m_1, Monad\ m_2, Monad\ m_3, MonadT\ t)\\ &\quad \Rightarrow (t\ m_2 \bowtie m_3) \to (m_1 \bowtie m_2) \to (t\ m_1 \bowtie m_3)\end{aligned}$$

so that, when composed from left to right, these masking views stack from top to bottom. For instance, Figure 8 depicts the mask $o \dotplus i \dotplus o$ that hides the first and third layer.

The masking and unmasking functionality is captured in the *to* and *from* fields of the $(m \bowtie n)$ record type

$$\begin{aligned}\mathbf{data}\ m \bowtie n = View\{&\ to \quad :: \forall a.m\ a \to n\ a\\ ,&\ from :: \forall a.n\ a \to m\ a\}\end{aligned}$$

For i masking and unmasking are simply the identity function.

$$\begin{aligned}i = View\{&\ to \quad = id\\ ,&\ from = id\}\end{aligned}$$

Ignoring the top-most layer with o means applying the monad zipper to shift down when masking, and up when unmasking.

$$\begin{aligned}o = View\{&\ to \quad = \downarrow\\ ,&\ from = \uparrow\}\end{aligned}$$

See that *to* and *from* are each other's inverses: $to\ v \circ from\ v \equiv id$ and $from\ v \circ to\ v \equiv id$. This is the prerequisite for *tmixmap*, which we use in the definition of a lifting function for views:

$$\begin{aligned}&vlift :: (MonadT\ t, Monad\ m, Monad\ n)\\ &\quad \Rightarrow (m \bowtie n) \to (t\ m \bowtie t\ n)\\ &vlift\ v = View\{\ to \quad = tmixmap\ (to\ v)\ (from\ v)\\ &\qquad\qquad ,\ from = tmixmap\ (from\ v)\ (to\ v)\}\end{aligned}$$

Lifting, together with horizontal composition

$$\begin{aligned}&(\dashv\vdash) :: (n \bowtie o) \to (m \bowtie n) \to (m \bowtie o)\\ &v_2 \dashv\vdash v_1 = View\{\ to \quad = to\ v_2 \circ to\ v_1\\ &\qquad\qquad ,\ from = from\ v_1 \circ from\ v_2\}\end{aligned}$$

are the two building blocks for vertical composition

$$v_2 \dotplus v_1 = v_2 \dashv\vdash (vlift\ v_1)$$

So $to\ (i \dotplus o)$ has type $t_1\ (t_2\ (t_3\ m))\ a \to t_1\ ((t_2 \triangleright t_3)\ m)\ a$: it hides the second layer.

4.2 Monad Masking in Action

Suppose that we modify our *Mem* feature to record how often a value is stored during evaluation for profiling purposes:

$$\begin{aligned}&\mathbf{type}\ Count = Int\\ &evalMem_2 :: (\mathbb{S}_M\ Reg\ (t\ m), \mathbb{S}_M\ Count\ m, MonadT\ t)\\ &\qquad \Rightarrow Open\ e\ Mem\ (t\ m\ Int)\\ &evalMem_2\ eval\ (Store\ e) =\\ &\quad \mathbf{do}\ count \leftarrow lift\ \$\ get\\ &\qquad lift\ \$\ put\ (count + 1)\\ &\qquad n \leftarrow eval\ e\\ &\qquad put\ n\\ &\qquad return\ n\\ &evalMem_2\ eval\ Retrieve = lift\ \$\ get\end{aligned}$$

Because the exception monad does not commute with the state monad, the relative position of the monad transformers in a stack affects the semantics. There are two possibilities:

- $\mathbb{E}_T\ e\ (\mathbb{S}_T\ s...)$: the state is preserved when an exception is thrown, or

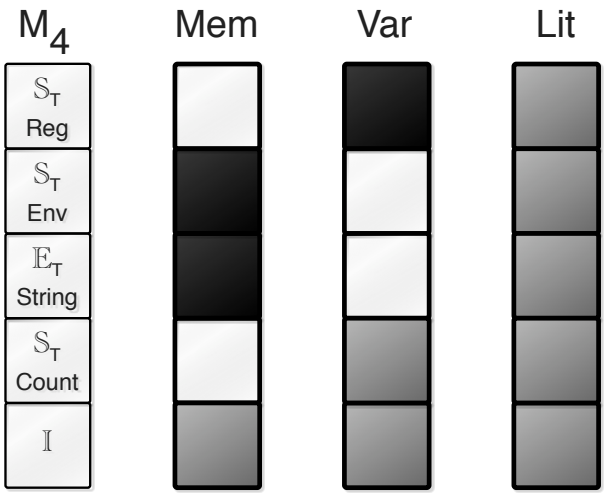

Figure 9. The monad stack (left) and the layers visible for the different features (right): black layers are masked and gray layers are abstracted over.

- $\mathbb{S}_T\ s\ (\mathbb{E}_T\ e...)$: the state is forgotten when an exception is thrown.

Hence, when assembling a language with variables and memory, we must carefully consider the monad stack configuration in terms of the desired semantics. A reasonable choice is, upon raising an exception, to forget the current variable environment and register, but to retain the profiling information.

$$\mathbf{type}\ M_4 = \mathbb{S}_T\ Reg\ (\mathbb{S}_T\ Env\ (\mathbb{E}_T\ String\ (\mathbb{S}_T\ Count\ \mathbb{I})))$$

So while $evalMem_2$ seemingly assumes consecutive state transformers, we actually wish to prise these appart, interleave them and wedge an exception transformer inbetween. The basic zipper approach of the previous section will not do that; $\otimes$ assumes the effects of one feature stay together. Fortunately, this is easily achieved with our masking language.

For convenience, the *fmask* combinator embeds a masked feature into an unmasked monad stack.

```
fmask :: (m ⋈ n) → Open e f (n a) → Open e f (m a)
fmask v eval_f eval = from v ∘ eval_f (to v ∘ eval)
```

Assembling the evaluation function, with features masked appropriately, becomes a breeze.

```
eval4 :: Expr3 → M4 Int
eval4 = fix ( fmask (i ⫩ o ⫩ o) evalMem2
            ⊛ fmask o evalVar
            ⊛ evalLit)
```

Figure 9 depicts the monad stack M_4 and the different masked views the features have of it.

5. The View Beyond the Mask

Views are not restricted to masking; they actually support arbitrary isomorphic *views* on the monad stack. In this generalized capacity, views resolve all kinds of compatibility issues between features. For instance, rather than to completely ignore or fully grant access to a transformer, a view can selectively disable functionality.

5.1 Read-Only View

We can restrict the read & write capabilities of a state monad transformer to read-only capability. For that purpose we exploit the representation of Monatron's reader transformer $\mathbb{R}_T$ as a newtype of $\mathbb{S}_T$

$$\mathbf{newtype}\ \mathbb{R}_T\ e\ m\ a = \mathbb{R}_T\{runR_T :: \mathbb{S}_T\ e\ m\ a\}$$

to get a very simple view

```
r :: S_T s m ⋈ R_T s m
r = View{ to   = R_T
        , from = runR_T }
```

Now we can express the evaluation of the Var feature with read-only access to the variable environment ($\mathbb{R}_M\ Env\ m$)

```
evalVar3 :: (R_M Env m, E_M String m)
         ⇒ Open e Var (m Int)
evalVar3 eval (Var n) =
  do env ← ask
     case lookup n env of
       Nothing → throw "Variable does not exist!"
       Just x  → return x
```

while, at the same time, the new post-increment feature has full access.

```
data Inc e = Inc String
evalInc :: (S_M Env m, E_M String m)
        ⇒ Open e Inc (m Int)
evalInc eval (Inc v) =
  do env ← get
     case lookup v env of
       Just n  → do put $ (v, n + 1) : delete (v, n) env
                    return n
       Nothing → throw "Variable does not exist!"
```

Using *fmask*, we combine both features with a shared state transformer.

```
type Expr4 = Fix (Var ⊕ Inc)
type M5    = S_T Env (E_T String I)
eval5 :: Expr4 → M5 Int
eval5 = fix ( fmask r evalVar3
            ⊛ evalInc)
```

More Views What other views can we express? The *write-only* view is the obvious companion of the read-only view. A state isomorphism can also be captured in a view

```
stateIso :: (s1 → s2) → (s2 → s1)
         → S_T s1 m ⋈ S_T s2 m
stateIso f f⁻¹ = View{ to   = iso f f⁻¹
                     , from = iso f⁻¹ f } where
  iso g h m = S_T $ λs2 → do (a, s1) ← runS_T (h s2) m
                             return (a, g s1)
```

which is useful for instance to share a register between features that expect values in different units (centimeters vs. inches or radians vs. degrees, say).

5.2 Internal Disambiguation with Explicit Views

The monad zipper resolves conflicts between *different* features with effects of the same type. However, the same issue arises when a single feature requires two different states, say. Obviously the *externally* applied zipper is no solution to this problem that already manifests itself inside the feature. The traditional solution to disambiguate two different states within a feature is to use *lift*.

Consider again the $evalMem_2$ example in Section 4.2. To disambiguate the $Count$ and Reg states, we have used *lift* for manipulating the former. Unfortunately, this internally motivated use of *lift* imposes a quite unnecessary ordering constraint: in the monad stack the Reg state transformer must appear above the $Count$ state transformer. Reversing the order or even overlapping (in suitable situations) the two transformers is not possible; for that purpose we need to change the feature implementation or write an alternate version.

Explicit view parameters allow us abstract from the ordering, similarly to Piponi (2010)'s *tagged transformers* but with two major advantages: *(i)* the existing transformer infrastructure does not have to be augmented with tags, and *(ii)* we get the full expressivity of views.

$$
\begin{aligned}
&evalMem_3 :: (\mathbb{S}_{\mathrm{M}}\ Reg\ m_1, \mathbb{S}_{\mathrm{M}}\ Count\ m_2, Monad\ m)\\
&\quad \Rightarrow (m \bowtie m_1) \rightarrow (m \bowtie m_2) \rightarrow Open\ e\ Mem\ (m\ Int)\\
&evalMem_3\ reg\ cnt\ eval\ (Store\ e) =\\
&\quad \mathbf{do}\ count \leftarrow get_v\ cnt\\
&\qquad put_v\ cnt\ (count + 1)\\
&\qquad n \leftarrow eval\ e\\
&\qquad put_v\ reg\ n\\
&\qquad return\ n\\
&evalMem_3\ reg\ cnt\ eval\ Retrieve = get_v\ reg
\end{aligned}
$$

where the auxiliary functions get_v and put_v apply the appropriate view. Doesn't that make the feature code look like *call-by-reference* of named variables *reg* and *cnt*?

$$
\begin{aligned}
&get_v :: \mathbb{S}_{\mathrm{M}}\ s\ n \Rightarrow (m \bowtie n) \rightarrow m\ s\\
&get_v\ var = from\ var\ get\\
&put_v :: \mathbb{S}_{\mathrm{M}}\ s\ n \Rightarrow (m \bowtie n) \rightarrow s \rightarrow m\ ()\\
&put_v\ var = from\ var \circ put
\end{aligned}
$$

The evaluation function $eval_{4b}$ is equivalent to $eval_4$, but uses the new $evalMem_3$. Note that we use the view i for *reg* to view the topmost state transformer, and o for cnt to see the other one.

$$
\begin{aligned}
&eval_{4b} :: Expr_3 \rightarrow M_4\ Int\\
&eval_{4b} = fix\ \ (\ fmask\ (i \dagger o \dagger o)\ (evalMem_3\ i\ o)\\
&\qquad\qquad \circledast fmask\ o\ evalVar\\
&\qquad\qquad \circledast evalLit)
\end{aligned}
$$

Changing the order of the two state transformers of $evalMem_3$ is as simple as swapping the two views.

$$
\begin{aligned}
&\mathbf{type}\ M_6 = \mathbb{S}_{\mathrm{T}}\ Count\ (\mathbb{S}_{\mathrm{T}}\ Env\ (\mathbb{E}_{\mathrm{T}}\ String\ (\mathbb{S}_{\mathrm{T}}\ Reg\ \mathbb{I})))\\
&eval_6 :: Expr_3 \rightarrow M_6\ Int\\
&eval_6 = fix\ \ (\ fmask\ (i \dagger o \dagger o)\ (evalMem_3\ o\ i)\\
&\qquad\qquad \circledast fmask\ o\ evalVar\\
&\qquad\qquad \circledast evalLit)
\end{aligned}
$$

6. Discussion

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

6.1 Monad Transformers and Data Types à la Carte

As shown in Section 2.5 extending modular features with effects is not straightforward. The techniques described in this paper can be viewed as an improvement of the existing techniques to support modular features *with* effects.

The approach of Liang et al. (1995) 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 (Swierstra 2008) avoids these hard references: it shows how to abstract away from the concrete compositions of datatypes (see Section 2.2). 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 2.5 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.

6.2 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 Liang et al. (1995). We have two main reasons to prefer Monatron over the MTL.

- The first reason is that in the MTL, the class for monad transfomers

  $$
  \begin{aligned}
  &\mathbf{class}\ MonadTrans\ t\ \mathbf{where}\\
  &\quad lift :: m\ a \rightarrow t\ m\ a
  \end{aligned}
  $$

  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 Jaskelioff (2009), 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 (Swierstra 2008). It would be also be interesting to adapt the zipper to other effect stacks such as *applicative functors* (Mcbride and Paterson 2008) or *arrows* (Hughes 1998).

7. Conclusion

With this pearl 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 of this pearl is currently available at `http://www.cs.kuleuven.be/~toms/Research/papers/MonadZipper.tgz`, but will shortly be released on Hackage as part of the Monatron library.

Acknowledgments

We are grateful to Jeremy Gibbons, Mauro Jaskelioff, Wonchan Lee, Wouter Swierstra and Stephanie Weirich for their help and feedback.

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.

References

W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

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

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

Mauro Jaskelioff. Monatron: An extensible monad transformer library. In *IFL '08: Symposium on Implementation and Application of Functional Languages*, 2008.

Mauro Jaskelioff. Modular monad transformers. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2009. Springer-Verlag.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, 1995.

Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

D. Piponi. Tagging monad transformer layers, 2010. `http://blog.sigfpe.com/2010/02/tagging-monad-transformer-layers.html`.

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