Monatron: An Extensible Monad Transformer Library

Mauro Jaskelioff

Functional Programming Laboratory
University of Nottingham

Abstract. Monads are pervasive in functional programming. In order to reap the benefits of their abstraction power, combinator libraries for monads are necessary. Monad transformers provide the basis for such libraries, and are based on a design that has proved to be successful. In this article, we show that this design has a number of shortcomings and provide a new design that builds on the strengths of the traditional design, but addresses its problems.

1 Introduction

The power of functional languages lies, to a great extent, in their ability to name and reuse programming idioms [6]. This power is often realised in the form of combinator libraries, which consist of a collection of idioms commonly found in the library's application area. Programmers can reuse these idioms and combine them to obtain programs which, since they are assembled from correct parts, are likely to be correct.

Monads [15, 18] are the standard abstraction for modelling programs with computational effects such as state, exceptions and continuations. These effects are usually associated with imperative languages but, with the help of monads, they can be elegantly modelled in a functional language [11]. However, obtaining a monad which models several combined effects can be difficult. Moreover, since monads must satisfy certain coherence conditions, the programmer is faced with the task of verifying these conditions. Obtaining combined effects can be made much easier with a good combinator library for monads. But, what are the properties that make a library good?

A good library should be *expressive*, in the sense that the exposed interface should be enough to obtain the desired combinations, without the need to look at the internals of the library. Since new applications might bring additional requirements, a library should be *extensible*. The semantics of the combinators should be *predictable* and, ideally, with no corner cases. These features help the programmer to abstract from low-level implementation details, and to think in terms of high-level idioms. Finally, a library should strive to be *efficient* and *portable*.

Combinator libraries for monads are built from modular components called monad transformers [14]. Current monad transformer libraries, such as mtl [4],

have been very successful in providing useful combinators for constructing monads. However, they have a number of shortcomings. Because the lifting of operations through monad transformers is done on a case-by-case basis, there is no guarantee that the liftings are uniform (Section 3.1) and it makes extending the library cumbersome (Section 3.2). Moreover, the lifting overloading mechanism produces shadowing of operations (Section 3.3), and relies essentially on non-portable features (Section 3.4).

The main contribution of this paper is the design of Monatron, a monad transformer library that addresses the issues discussed above (Section 4). Its implementation (Section 5) builds on the strengths of existing monad transformer libraries and incorporates some new ideas on liftings of operations [9] that are based on solid theoretical principles. Although these lifting of operations have been implemented by the author in the library mmtl[1], the implementation of mmtl closely followed the design in existing libraries and, as consequence, still suffered from some of their problems. The desire to eliminate these problems motivated the design of the Monatron [8] library.

The organization of the paper is as follows: in Section 2 we explain how to construct complex monads using monad transformers. In Section 3 we explain some issues raised by the current design, and in Section 4 we introduce a new design that addresses these issues. The implementation of the design is carried out in Section 5 and in Section 6 we conclude.

2 Combining Monads

Monads provide an extremely useful abstraction for computational effects. In Haskell, monads are given by instances of the following type class:

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

Intuitively, m is a type constructor of computations, return introduces a trivial computation, and (>>=) (pronounced *bind*) sequences computations by taking a computation returning a value of type a and a function from a into a new computation. Instances of the Monad class are required to satisfy certain coherence equations that ensure that return and (>>=) are well-behaved.

As an example, in Figure 1 we have defined two monads: Either x is a monad for exceptions of type x, and the identity monad Id is a monad of pure computations.

Combinator libraries for monads come equipped with some standard monads corresponding to different computational effects that provide readily available building blocks for constructing effectful computations. For example, libraries usually provide a State s monad for modelling global state of type s, a Cont r monad for modelling continuations with result type r, a Writer w monad for

[1] http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mmtl

```
data Either x a = Left x | Right a
instance Monad (Either x) where
  return a       = Right a
  Left x  >>= f  = Left x
  Right a >>= f  = f a
```

```
newtype Id a = Id a
instance Monad Id where
  return a      = Id a
  (Id a) >>= f  = f a
```

Fig. 1. Either and Identity Monad

modelling output of a monoid w, a Reader e monad for modelling environments of type e, and an Exception x monad for modelling exceptions of type x. To see the details of their implementation we refer the reader to an introductory article [1]. These monads provide some of the most common computational effects but, by all means, they are not the only ones. The fact that monad libraries (mostly) only support this limited set of effects and have not been extended to other effects can be seen as evidence of the universality of these effects. Less optimistically, it can be seen as symptomatic of a lack of extensibility (Section 3.2).

In addition to the aforementioned monads, libraries provide the corresponding monad transformer [14] for each effect. A *monad transformer* is a monad parameterised by an arbitrary monad. It can be thought of as a monad with a hole which can be filled by any monad. More formally, a monad transformer is a type constructor with kind $(* \to *) \to (* \to *)$ which is an instance of the type class MonadT.

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

Each instance T of MonadT is required to map monads to monads (via an instance Monad m ⇒ Monad (T m)), and lift is required to behave well with respect to return and (>>=).

Using monad transformers, one can easily construct complex monads. For example, the state monad transformer StateT s applied to some monad m will yield a monad StateT s m, which combines the effect of state and the effect(s) modelled by m. One can then stack effects, adding more and more monad transformer layers, in order to construct complex monads. For example, **type** StExc s x = StateT s (Exception x) is a monad for state of type s and exceptions of type x. We can further extend this monad and add continuations simply by applying the continuation monad transformer to obtain **type** ContStExc r s x = ContT r (StExc s x).

Since a monad transformer adds an effect to another monad, monads such as State can be constructed by adding state to the identity monad Id.

$$\mathsf{State}\ s \equiv \mathsf{StateT}\ s\ \mathsf{Id}$$

Therefore, in principle, only the transformer version of the monad is needed. Also, note that for the combined monads there is no need to verify any monad laws, or define new monad instances. The monad transformers will guarantee

that the obtained type is a monad by construction, realising the idea that correct constructions are obtained by combining correct components.

Consider now the monad **type** ExcSt $x\ s$ = ExcT x (State s), obtained by applying the exception monad transformer to the state monad. Is this monad equivalent to StExc $s\ x$? After all, both monads model state together with exceptions. The answer is no, and in general, the order in which the monad transformers are applied is important. To see why, it is necessary to *run* the effectful computations.

Monad transformers and their associated monads come equipped with a run function that allows us to evaluate an effectful computation. For example, the state monad can be run with runState :: $s \to$ State $s\ a \to (a, s)$ that given an initial state and a stateful computation, returns the value of the computation together with the final state. The state monad transformer can be run with runStateT :: Monad $m \Rightarrow s \to$ StateT $s\ m\ a \to m\ (a, s)$, which given an initial state and a computation, returns an m-computation with a value and final state. The exception monad[2] is run with a function runException :: Exception $x\ a \to$ Either $x\ a$, and the exception monad transformer is run with a function runExcT:: Monad $m \Rightarrow$ ExcT $x\ m\ a \to m$ (Either $x\ a$), that returns an m-computation over either an exception or a value.

In order to obtain a run function for a combined monad we compose the run functions of the components:

```
runStExc  :: s → StExc s x a → Either x (a, s)
runStExc s = runException · runStateT s

runExcSt  :: s → ExcSt x s a → (Either x a, s)
runExcSt s = runState s · runExcT
```

Analysing the type of the resulting run functions, we can see that in StExc when an exception is raised the computation forgets about the state, while in ExcSt when an exception is raised the computation preserves the state. One can then choose how exceptions should interact with state by choosing the order in which the monad transformers are applied. In general, applying monad transformers in different orders gives rise to different interactions among effects [14].

So far, we have discussed the construction of types with monadic structure for modelling computational effects. In order to write programs that will produce and manipulate these effects, monads and monad transformers come equipped with effectful operations. For example, State comes equipped with operations get :: State $s\ s$ and put :: $s \to$ State s () for getting access to the current state and for setting the current state, and Exception comes equipped with operations throw :: $x \to$ Exception $x\ a$ and handle :: Exception $x\ a \to (x \to$ Exception $x\ a) \to$ Exception $x\ a$ for throwing an exception and handling an exception.

Consider the exception monad Exception x which supports the operation throw. If we extend it with a monad transformer such as StateT s and obtain

[2] We distinguish between the Either monad and the exception monad as they might have different implementations. Also, it is precisely when they differ that runException is interesting.

the monad StateT s (Exception x), we expect this extended monad to not only support get and put as provided by StateT, but also to support throw as provided by Exception x. How to make the extended monad to support the operations of its underlying monad is known as the problem of *operation lifting*.

3 Some Problems with the Traditional Design

The current design of monad transformers libraries performs the liftings of operations of an underlying monad to the transformed monad in an ad-hoc fashion, relying crucially on a type-class trick in order to perform the liftings. The basic idea is the following: define a class of monads supporting a certain operation. For example, we define the class of monads supporting the operation callCC, and show that the continuation monad transformer ContT applied to any monad m is an instance of this class:

```
class Monad m ⇒ ContM m where
  callCC :: ((a → m b) → m a) → m a
instance Monad m ⇒ ContM (ContT m) where
  callCC = ...
```

The final step is to show that, for each monad transformer in the library, if the underlying monad supports callCC, then the transformed monad also supports callCC. For example, for the exception monad transformer ExcT:

```
instance ContM m ⇒ ContM (ExcT x m) where
  callCC = ...
```

This type-class trick has some advantages, such as an overloading of the operations that is usually convenient, but libraries that rely on it have some shortcomings which affect the predictability, extensibility, expressive power and portability of the library. In the following, we explain why this is so.

3.1 Non-uniform liftings

One can replace a computation on the Writer monad by a computation on the more general State monad, and replace the output operation tell by an output operation on State. Similarly, one can do the same replacements on their transformer versions, for example, replacing the operation tell :: Monad m ⇒ String → WriterT String m () that outputs a string, with the following definition:

```
tell^S   :: Monad m ⇒ String → StateT String m ()
tell^S w = do s ← get
              put (s ++ w)
```

One would expect that replacing WriterT by StateT, replacing tell by tell^S, and replacing runWriterT by runStateT "", the semantics of a program would be preserved. However, in the mtl [4], the following two programs, which perform

computations over WriterT String Cont and StateT String Cont respectively, have different behaviours:

```
p1 :: (String, String)
p1 = (runCont id · runWriterT)
       (callCC (λexit → tell "1" >>= λ_ → exit "Exit"))
p2 :: (String, String)
p2 = (runCont id · runStateT "")
       (callCC (λexit → tell^S "1" >>= λ_ → exit "Exit"))
```

While p1 = ("Exit", ""), we have that p2 = ("Exit", "1"). The difference in behaviour is caused by non-uniform liftings in the mtl. In particular, callCC is lifted through the StateT monad transformer in a way which is not coherent with the lifting of callCC through WriterT. Although we can regard this as a bug and change the implementation of the library, since each operation is lifted on a case-by-case basis, there is no intrinsic guarantee that the liftings are coherent. Hence, with no guarantee that the liftings are coherent, the predictability of the semantics of the library is seriously affected.

3.2 Quadratic number of instances

Suppose a programmer wants to extend the library with a new monad transformer which comes equipped with some operations. The programmer must write a new class corresponding to the added operations and an instance of this new class for *each* existing monad transformer, so that the added operations can be lifted through other monad transformers. Furthermore, the programmer is required to write instances of *each* class of existing operations for the new monad transformer. In other words, assuming one class of operations per monad transformer, the number of instances increases quadratically with the number of monad transformers.

Not only the extensibility of the library is affected because of the quadratic growth of required lines of code, but also because of the lack of separation of concerns. Extending the library requires understanding the semantics of all the existing monad transformers.

The quadratic number of instances and lack of separation of concerns is a major hurdle. It discourages anyone willing to extend the library and it shows that the traditional design can only work for a library with a very limited number of monads.

3.3 Shadowing of operations

We have seen in Section 2 how StExc and ExcST give rise to different interactions between exceptions and state. With the former, state changes are lost when an exception is raised, while with the latter state changes are preserved. Suppose that we need both types of exception. We can easily construct such a monad as follows:

```
type ExcStExc x1 s x2 = ExcT x1 (StExc s x2)
runExcStExc :: s → ExcStExc x1 s x2 a → Either x2 (Either x1 a, s)
runExcStExc s = runStExc s · runExcT
```

We now have two different types of exceptions with two different exception objects (*x1* and *x2*). Let us assume that both objects are of the same type, say Int. Since there is no type that will distinguish instances, the function handle will refer to the instance of the outermost monad. We have no way of handling the other type of exceptions, as there is no way of saying "What I mean is the handle operation corresponding to the monad under two monad transformers". The inner handle operation is shadowed by the outer one.

One way to deal with this problem is to define different types of exceptions, but this usually means inserting unnecessary constructors and destructors which clutter the program and make it more difficult to understand. We see shadowing as revealing a limitation in the expressive power of the library.

3.4 Portability

The implementation of the type-class trick requires several extensions to the Haskell 98 standard [12], such as functional dependencies and multi-parameter classes. Although many of these extensions have been around for a while and their semantics are quite stable, they certainly affect the portability of the library. This is particularly so because the whole implementation of the library revolves around this type-class trick. We would like to have the *choice* of paying the price and using the operation overloading provided by the type-class implementation when it is convenient without being *forced* to do so.

4 A New Approach

We now present the new ideas behind our monad transformer library. Specifically, we present an improved lifting mechanism with explicit lifting of functions which addresses the significant shortcomings discussed in the previous section.

The main idea of this approach is to pair each operation to a type of its implementations. For example, implementations of handle for exceptions of type x and a monad m are given by elements of the type Handle x m. Once a type of implementations is given, we define functions parameterised by an implementation. In the exceptions handling example, this means that we will define a function handleX that given an implementation Handle x m performs the handling of an exception.

```
handleX :: Monad m ⇒ Handle x m → m a → (x → m a) → m a
handleX = ...
```

Additionally, we define a function liftHandle that, given an implementation of the operations for a monad m, produces an implementation of the operation for the monad t m, where t is an arbitrary monad transformer (as indicated by the class constraint MonadT t).

```
liftHandle :: (Monad m, MonadT t) ⇒ Handle x m → Handle x (t m)
liftHandle = ...
```

We expect every operation to have a type for its implementations, and a function parameterised by this type which allows programmers to use the operation and a lifting function.

Concrete monads and monad transformers provide their own implementation of the operations:

```
handleExcT :: Monad m ⇒ Handle x (ExcT x m)
handleExcT = ...
```

It is now possible, when using an operation, to state explicitly and precisely which implementation is meant and through how many monad transformers this implementation should be lifted. For example, handleX handleExcT is the operation handle as implemented by handleExcT. Its lifting through two monad transformers is given by handleX (liftHandle (liftHandle handleExcT)).

The approach has several advantages:

Uniform-liftings: operations are lifted uniformly through monad transformers by construction. This means that the semantics of the lifted operations is predictable.

Modularity: operations need to be defined only once for each monad/monad transformer that supports them, effectively taking the quadratic order of instances to linear.

Expressivity: One has the ability to exactly state which operation one is referring to, as it is done in the example above. In fact, if desired, one can have more that one implementation of an operation for a given monad.

The main difference between the traditional approach and ours is that instead of defining a type class for each class of operations, we define a type. This removes the dependency from type classes, but opens the question of what is an appropriate type for the implementation of operations. We answer this question in Section 5.

The implementation that follows the ideas above constitutes the *core* of our library Monatron. This core only needs Haskell 98 extended with rank-2 types [10] and is fully-functional. However, the overloading of operations provided by type-classes is often convenient. When there is no possible confusion, and using additional language extensions is not a problem, one can let the compiler infer which implementation one is referring to.

4.1 Overloading of operations

It is simple to add overloading on top of the core functionality. We define the class of monads m with implementations Handle x m, and define the operation handle for this class of monads:

```
class Monad m ⇒ HandleM x m | m → x where
  handleM :: Handle x m
handle :: HandleM x m ⇒ m a → (x → m a) → m a
handle = handleX handleM
```

Note that in the definition of handle, we are replacing the explicit parameter Handle z m in handleX by an implicit parameter given by the type class.

Finally, we provide instances for the concrete monads that implement the operations: the monad ExcT x m, and any monad transformer applied to a monad of class HandleM.

```
instance Monad m ⇒ HandleM x (ExcT x m) where
  handleM = handleExcT
instance (HandleM x m, MonadT t) ⇒ HandleM x (t m) where
  handleM = liftHandle handleM
```

Of course, implementing this overloading of operations will require the use of many extensions to Haskell 98, as it was the case with the traditional design of monad transformer libraries. However, in this case, the core functionality does not rely on the extensions and one has the choice of using the extensions when they are available and overloading is convenient, as opposed to being forced to always do so.

5 Implementation of Monatron

The key idea of the design is to obtain a suitable representation of the implementations of the operations. Such a representation should allow programmers to both perform the operations and lift them through monad transformers.

5.1 Types for operation implementation

Let us consider first the simple case of the operations get and put. A monad m implements these operations for a parameter s with a pair of a computation $m\ s$ and a function $s \to m\ ()$. Hence, our representation for the operations is a pair as shown in the type GetPut below. The projections give rise to the parameterised operations getX and putX, and lifting an implementation is just post-composing lift.

```
type GetPut s m = (m s, s → m ())
getX          :: GetPut s m → m s
getX (g, _)   = g
putX          :: GetPut s m → s → m ()
putX (_, p) s = p s
liftGetPut        :: (Monad m, MonadT t) ⇒ GetPut s m → GetPut s (t m)
liftGetPut (g, p) = (lift g, lift · p)
```

```
newtype Cod f a = Cod {unCod :: ∀b. (a → f b) → f b}
runCod :: Monad m ⇒ Cod m a → m a
runCod c = unCod c return
instance MonadT Cod where
  lift m     = Cod (m >>=)
  tmap f m = Cod (λk → f (unCod m (f · k)))
instance Monad m ⇒ Monad (Cod m) where
  return     = lift · return
  c >>= f    = Cod (λk → unCod c (λa → unCod (f a) k))
```

Fig. 2. Codensity monad transformer

The function liftGetPut is simple because in the type of the operations of GetPut the monad only occurs in the result. In general, operations of the form $\forall\, a.\ \mathsf{S}\ a \to \mathsf{M}\ a$, where S is an instance of Functor, can be lifted simply by applying lift after the operation. Operations of the form above are in one-to-one correspondence with operations $\forall x.\ \mathsf{S}\ (\mathsf{M}\ a) \to \mathsf{M}\ a$ that respect the bind of the monad (i.e. operations such that it is the same to bind any function $a \to \mathsf{M}\ b$ before performing the operation or after performing the operation). Because of this correspondence we will say that such operations are S-algebraic.

What happens when the operation in question is not S-algebraic? For example, the operation for handling exceptions of type x has type $\forall\, a.\ \mathsf{M}\ a \to (x \to \mathsf{M}\ a) \to \mathsf{M}\ a$, for a monad M which supports exceptions. Although it is equivalent to an operation $\forall a.\ \mathsf{S}\ (\mathsf{M}\ a) \to \mathsf{M}\ a$ for $\mathsf{S}\ a = (a, x \to a)$, it is not S-algebraic, since it doesn't respect the bind of the monad (handling an exception before binding and after binding an arbitrary function $a \to \mathsf{M}\ b$ does not yield the same result).

However, it can be shown that every operation $\mathsf{op} :: \forall\, a.\ \mathsf{S}\ (\mathsf{M}\ a) \to \mathsf{M}\ a$ is in one-to-one correspondence with an operation $\overline{\mathsf{op}} :: \forall\, a.\ \mathsf{S}\ a \to \mathsf{Cod}\ \mathsf{M}\ a$. Here, Cod M a is the codensity monad transformer [13, 9] defined in Fig. 2. Categorically, the codensity monad transformer takes a functor to the right Kan extension of the functor along itself. The correspondence between op and $\overline{\mathsf{op}}$ shows how an operation which is not S-algebraic for a given monad can be S-algebraic for a different (more powerful) monad.

As with any monad transformer, we can lift a computation ($m\ a$) into (Cod $m\ a$) with lift, but also we can use runCod to run a computation (Cod $m\ a$) and obtain a computation ($m\ a$) in the original monad. A routine calculation shows that $\mathsf{runCod} \cdot \mathsf{lift} = \mathsf{id}$.

The obtained operation $\overline{\mathsf{op}}$ can be easily lifted to a monad transformer T by post-composition with lift. However, this does not yield a lifting of op yet. In order to obtain a proper lifting, we need to 'correct' the result of the operation ($\mathsf{lift} \cdot \overline{\mathsf{op}}$), by applying ($\mathsf{lift} \cdot \mathsf{runCod}$) to every occurrence of the monad inside the transformer T. Hence, we extend the class for monad transformers with an extra member:

```
type Handle x m = (∀a. a → (x → a) → m a, ∀a. m a → m a)
handleX          :: Monad m ⇒ Handle x m → m a → (x → m a) → m a
handleX (h, n) m = n · join · h m
liftHandle       :: (Monad m, MonadT t) ⇒ Handle x m → Handle x (t m)
liftHandle (h, n) = (λm → lift · h m, tmap n)
```

Fig. 3. Type and functions for implementations of handle.

```
type Throw x m = (∀a. x → m a)
throwX          :: Throw x m → x → m a
throwX t x      = t x
liftThrow       :: (Monad m, MonadT t) ⇒ Throw x m → Throw x (t m)
liftThrow t     = lift · t
```

Fig. 4. Type and functions for implementations of throw.

```
class MonadT t where
  lift :: Monad m ⇒ m a → t m a
  tmap :: (Monad m, MonadT t) ⇒ (∀x. m x → m x) → t m a → t m a
```

The addition of tmap is essential for lifting the correction function, not only for the case of handle, but also for any other operation which is not S-algebraic.

If we lift an operation through two transformers the correction function is no longer (lift · runCod) :: M a → M a, but tmap (lift · runCod) :: T M a → T M a. Consequently, the correction function will change according to the depth of the lifting. This motivates the following definition of the implementation of handle.

Implementations of handle consist of pairs of an S-algebraic operation and a correction function (see Figure 3). The parameterised operation handleX constructs from these components the usual operation exception handling. The function liftHandle lifts the components with lift and tmap respectively.

As a final example, in Figure 4 we provide the type and functions for implementations of throw. Since throw is an S-algebraic operation, its implementations are simply functions of type $\forall a.\ x \to m\ a$.

5.2 Implementing Transformers and Operations

So far, we have defined functions that deal with generic implementations of certain operations. For example, handleX and liftHandle deal with implementations Handle x m. We now define concrete monad transformers and show that they provide an implementation of operations. In particular, we will define the state monad transformer and its implementation of get and put, and the exception monad transformer and its implementation of throw and handle.

State Monad Transformer We start with the state monad transformer (see Figure 5). Our implementation of the state monad transformer is essentially the traditional one [14].

```
newtype StateT s m a = S {unS :: s → m (a, s)}
stateT :: (s → m (a, s)) → StateT s m a
stateT = S
runStateT :: s → StateT s m a → m (a, s)
runStateT s m = unS m s
instance MonadT (StateT s) where
  lift m = S (λs → m >>= λa → return (a, s))
  tmap f (S m) = S (f · m)
instance Monad m ⇒ Monad (StateT s m) where
  return   = lift · return
  m >>= k = S (λs → unS m s >>= λ(a, s') → unS (k a) s')
```

Fig. 5. State monad transformer

Note that we give a different name for the constructor and destructor. This is because we only want to export an interface consisting of a type and some operations, leaving room for potential optimizations.

The monad transformer StateT provides an implementation of GetPut, which is simply a pair of functions.

```
getPutStateT :: Monad m ⇒ GetPut s (StateT s m)
getPutStateT = (get, put)
  where get _ = S (λs → return (s, s))
        put s = S (λ_ → return ((), s))
```

Once we have defined the implementation of GetPut, we can use it via getX and putX, and lift it with liftGetPut. Importantly, there is a separation of the notion of *implementation* of operations (as provided by getPutStateT) and the *use* of that type of operation (as provided by the functions getX, putX, and liftGetPut).

Exception Monad Transformer Let us review the traditional definition of the exception monad transformer, which can be found in Figure 6, together with the functions throw^i and handle^i. A computation of type $\mathsf{X}^i\ x\ m\ a$ is an m-computation whose value is either an exception of type x or a pure value of type a. The operation throw^i throws an exception, and handle^i takes a computation and a handler, and if the computation throws an exception then it applies the handler.

This monad transformer can provide an implementation $\mathsf{Throw}\ x\ (\mathsf{X}^i\ x\ m)$, but it is not powerful enough to provide an implementation $\mathsf{Handle}\ x\ (\mathsf{X}^i\ x\ m)$, as we would have no way of providing a lifting function for it.

Consider now a context $[\bullet] \gg\!= k$ that binds a function $k :: a \to \mathsf{X}^i\ x\ m\ b$. Given such a context, it is possible to define an exception handling operation which instead of checking if its first argument throws an exception, it checks whether an exception is thrown after binding k:

```
newtype X^i x m a = X^i {unX^i :: m (Either x a)}
instance MonadT (X^i x) where lift m   = X^i (fmap Right m)
                              tmap f = X^i . f . unX^i
instance Monad m => Monad (X^i x m) where
  return        = lift . return
  (X^i m) >>= f = X^i (do a <- m; case a of Left x  -> return (Left x)
                                            Right b -> unX^i (f b))
throw^i       :: Monad m => x -> X^i x m a
throw^i x     = X^i (return (Left x))
handle^i      :: Monad m => X^i x m a -> (x -> X^i x m a) -> X^i x m a
handle^i m h = X^i (unX^i m >>= \exa -> case exa of
                                          Left x  -> unX^i (h x)
                                          Right a -> return (Right a))
```

Fig. 6. Traditional exception monad transformer

```
newtype ExcT x m a = X {unX :: Cod (X^i x m) a}
instance MonadT (ExcT x) where
  lift      = X . lift . lift
  tmap f = X . tmap (tmap f) . unX
instance Monad m => Monad (ExcT x m) where
  return   = lift . return
  m >>= f = X (unX m >>= unX . f)
```

Fig. 7. Exception monad transformer

$$h = \mathsf{handle}^i\ (k\ a)\ (k \cdot h) :: a \to (x \to a) \to \mathsf{X}^i\ x\ m\ a$$

Analyzing computations in the context of a bind operation is possible if we transform $\mathsf{X}^i\ x\ m$ with the codensity monad transformer. Intuitively, the codensity monad transformer adds to a monad the possibility of analysing a computation in the context of a bind.

The proposed correction function ($\mathsf{lift} \cdot \mathsf{runCod}$) has the effect of closing the context. In order to illustrate this idea, consider the computation $\mathsf{handle}\ m\ h \gg\!= \lambda\_ \to \mathsf{throw}\ ()$. We do not expect the throw after the bind to affect the result of the handle operation. This means that, even though we are considering computations in a context, at some point we want to "close" the context, perform a computation, and only then consider the result in a resulting context. This is exactly what ($\mathsf{lift} \cdot \mathsf{runCod}$) does.

We can define the exception monad transformer as the transformation of the traditional exception monad with the codensity monad transformer. The complete implementation of the obtained monad transformer for exceptions is shown in Figure 7.

We remark that, despite the relative complexity of the obtained monad, the interface for it does not need to be changed. It is still possible to construct elements of the exception monad transformer $\mathsf{ExcT}\ x\ m\ a$ from an element of $m\ (\mathsf{Either}\ x\ a)$, and running the exception monad still yields the type $m\ (\mathsf{Either}\ x\ a)$, as shown below:

$$\begin{aligned}
&\mathsf{excT} :: \mathsf{Monad}\ m \Rightarrow m\ (\mathsf{Either}\ x\ a) \to \mathsf{ExcT}\ x\ m\ a\\
&\mathsf{excT} = \mathsf{X} \cdot \mathsf{lift} \cdot \mathsf{X}^i\\
&\mathsf{runExcT} :: \mathsf{Monad}\ m \Rightarrow \mathsf{ExcT}\ x\ m\ a \to m\ (\mathsf{Either}\ x\ a)\\
&\mathsf{runExcT} = \mathsf{unX}^i \cdot \mathsf{runCod} \cdot \mathsf{unX}
\end{aligned}$$

This means that the complexity of the implementation is actually hidden from the programmer, who can use the library without needing to understand the details of the implementation.

Going to the trouble of defining the exception monad transformer in this manner now pays off. We can now define the implementation of the operations throw and handle as provided by ExcT. Note that the correction function in the implementation of handle is essentially $(\mathsf{lift} \cdot \mathsf{runCod})$.

$$\begin{aligned}
&\mathsf{throwExcT} :: \mathsf{Monad}\ m \Rightarrow \mathsf{Throw}\ x\ (\mathsf{ExcT}\ x\ m)\\
&\mathsf{throwExcT}\ x = \mathsf{X}\ (\mathsf{lift}\ (\mathsf{throw}^i\ x))\\
&\mathsf{handleExcT} :: \mathsf{Monad}\ m \Rightarrow \mathsf{Handle}\ x\ (\mathsf{ExcT}\ x\ m)\\
&\mathsf{handleExcT} = (k, \mathsf{excT} \cdot \mathsf{runExcT})\\
&\quad \textbf{where}\ k\ a\ h = \mathsf{X}\ (\mathsf{Cod}\ (\lambda c \to \mathsf{handle}^i\ (c\ a)\ (c \cdot h)))
\end{aligned}$$

The given semantics for handleExcT is the expected one, as a routine calculation shows that, in essence, it is equivalent to handle^i:

$$\mathsf{handleX}\ \mathsf{handleExcT} \equiv \mathsf{handleE}^i$$

where $\mathsf{handleE}^i$ is given by:

$$\begin{aligned}
&\mathsf{handleE}^i :: (\mathsf{Monad}\ m) \Rightarrow \mathsf{ExcT}\ x\ m\ a \to (x \to \mathsf{ExcT}\ x\ m\ a) \to \mathsf{ExcT}\ x\ m\ a\\
&\mathsf{handleE}^i\ m\ h = \mathsf{xe}\ (\mathsf{handle}^i\ (\mathsf{ex}\ m)\ (\mathsf{ex} \cdot h))\\
&\quad \textbf{where}\ \mathsf{xe} :: \mathsf{Monad}\ m \Rightarrow \mathsf{X}^i\ x\ m\ a \to \mathsf{ExcT}\ x\ m\ a\\
&\qquad\quad \mathsf{xe} = \mathsf{X} \cdot \mathsf{lift}\\
&\qquad\quad \mathsf{ex} :: \mathsf{Monad}\ m \Rightarrow \mathsf{ExcT}\ x\ m\ a \to \mathsf{X}^i\ x\ m\ a\\
&\qquad\quad \mathsf{ex} = \mathsf{runCod} \cdot \mathsf{unX}
\end{aligned}$$

Once that the implementation of the operations is defined, we are done. Using and lifting the operations can be done by the functions that act on arbitrary implementations of the operations (such as those in Figure 3).

6 Conclusion

Combinator libraries for monads are essential for facilitating the construction of complex monads that naturally appear in applications that go from basic parser

libraries [7] to end-user applications [16]. We have shown that the current design of monad transformer libraries has a number of shortcomings that hinder the extensibility, predictability, portability, and expressive power of the library.

By restructuring the design and incorporating uniform liftings of operations we have managed to address all of these issues, except for portability for which, at least, some alternative is given. This new design guided the implementation of the Monatron library [8]. The design requires a deeper understanding from the monad transformer writer, as operations need to be defined in such a way that they can be lifted through arbitrary monad transformers. One way to obtain these liftings is formally explained using system $F\omega$ in [9]. The user of the library sees none of this complexity, and benefits from a more expressive interface that works uniformly on all operations.

The complexities of the design lead us to hide the internal implementation in order to maintain a simple interface. This was unnecessary before, as the implementation was very close to the interface. This change may prove to be useful for improving the efficiency of the library, as hidden implementations allow for optimizations that preserve the interface. For example, the list monad is often used for modelling non-determinism, but its *merge* operation (concatenation) is rather inefficient. Using a different internal structure, but preserving the interface, we can provide an efficient *merge* operation. We leave as future work a further departure from the traditional implementation of monads in search for better performance.

6.1 Related Work

The mtl [4] is the most well-known monad transformer library. It is inspired by the paper of Liang et al. [14], and for many years it has been distributed together with the Haskell compiler GHC. More recently, a new library called MonadLib [3] has been introduced. This library is an improvement over the mtl, but it still suffers from the problems described in Section 3. The library presented in this article owes a lot to the excellent work done by the authors of these two libraries.

The codensity monad transformer has appeared in a number of functional programming papers. For example, it has been derived as a monad transformer for backtracking [5], it has been calculated in a search for efficient parsers [2], and it has been used to optimize substitution in the free monad [17]. In our case, however, we were motivated by its mathematical properties [9].

Acknowledgements. I would like to thank Graham Hutton and the anonymous referees for their comments and suggestions and all the people in the Functional Programming Lab for their support.

References

1. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School On Applied Semantics APPSEM2000*, pages 42–122. Springer-Verlag, 2000.

2. K. Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.
3. Iavor Diatchki. Monadlib. `http://www.galois.com/~diatchki/monadLib/`.
4. A. Gill. Monad transformer library. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mtl-1.1.0.2`.
5. R. Hinze. Deriving backtracking monad transformers. In *ICFP*, pages 186–197, 2000.
6. J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *AFP*, pages 53–96. Springer Verlag, LNCS 925, 1995.
7. G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.
8. M. Jaskelioff. Monatron. `http://www.cs.nott.ac.uk/~mjj/monatron`.
9. M. Jaskelioff. Modular monad transformers. In *ESOP*, 2009. Accepted for publication.
10. M. P. Jones. First-class polymorphism with type inference. In *POPL*, pages 483–496. ACM Press, 1997.
11. S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.
12. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
13. S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971. Second edition, 1998.
14. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343, 1995.
15. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
16. D. Stewart and S. Sjanssen. Xmonad. In *Haskell Workshop*, pages 119–119, New York, NY, USA, 2007. ACM.
17. J. Voigtländer. Asymptotic improvement of computations over free monads. In *MPC*, pages 388–403, 2008.
18. P. Wadler. The essence of functional programming. In *POPL*, pages 1–14, 1992.