Monad Transformers Step by Step

Martin Grabmüller

Oct 16 2006 (Draft)*

Abstract

In this tutorial, we describe how to use monad transformers in order to incrementally add functionality to Haskell programs. It is not a paper about *implementing* transformers, but about *using* them to write elegant, clean and powerful programs in Haskell. Starting from an evaluation function for simple expressions, we convert it to monadic style and incrementally add error handling, environment passing, state, logging and input/output by composing monad transformers.

1 Introduction

This paper gives a step-by-step introduction to monad transformers in Haskell.

Monads are a remarkably elegant way for structuring programs in a flexible and extensible way. They are especially interesting in a lazy functional language like Haskell, because they allow the integration of side-effects into otherwise purely functional programs. Furthermore, by structuring a program with monads, it is possible to hide much of the necessary book-keeping and plumbing necessary for many algorithms in a handful of definitions specific for the monad in use, removing the clutter from the main algorithm.

Monad transformers offer an additional benefit to monadic programming: by providing a library of different monads and types and functions for combining these monads, it is possible to create custom monads simply by composing the necessary monad transformers. For example, if you need a monad with state and error handling, just take the *StateT* and *ErrorT* monad transformers and combine them. The goal of this paper is to give a gentle introduction to the use of monad transformers by starting with a simple function and extending it step by step with various monadic operations in order to extend its functionality. This paper is *not* about the theory underlying the monad transformer concept, and not about their implementation (except for what is necessary for successfully using them).

The reader is expected to be familiar with functional and basic monadic programming, such as the use of the *Monad* class and **do** notation in Haskell. Everything else will be explained on the fly.

The Haskell programs in this paper use language features which are not in the current Haskell'98 standard, since they use the non-standard library modules *Control.Monad.Error* etc. Both the hierarchical module names of these modules and some of their implementation details are beyond Haskell'98. Nevertheless, these extensions are well supported in current versions of the Glasgow Haskell compiler (GHC) [**?**] and the Hugs Haskell interpreter [**?**]. The programs have been tested using versions 6.4.2 and 6.5 (pre-release) of GHC and version 20050308 of Hugs[1].

The monad transformer modules are inspired by a paper by Mark P. Jones [**?**], which gives a very readable introduction to monadic programming, but is less practical than this paper.

This document has been converted from a literate Haskell script using Andres Löh's `lhs2TeX`[2]

*Sections **??** and **??** are added by Wei Hu, Feb 2008. Unfortunately the bib file is missing, so are the references.

[1] You need to pass the `-98` option to Hugs in order to allow the required Haskell extensions.

[2] `http://www.iai.uni-bonn.de/~loeh/`

preprocessor. The script is executable by both GHC and Hugs. The literate Haskell source file `Transformers.lhs` is available from my homepage[3][4].

It is probably best to read this paper near a computer so that you can look up the types and descriptions of the various functions used from the monad transformer library or the Haskell standard library. The best setup is a printed copy of this paper, a web browser showing the online library documentation and a running incarnation of ghci or hugs with a loaded *Transformers* module (to be introduced below) for checking out types interactively using the `:type` (or `:t`) command.

1.1 Example Program

As a running example, an interpreter for a simple programming language will be used throughout the paper. All the code will be located in a module called *Transformers*, which has the following header:

```
module Transformers where
import Control.Monad.Identity
import Control.Monad.Error
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer
import Data.Maybe
import qualified Data.Map as Map
```

Several of the imported modules beginning with *Control.Monad* are only needed when you use the monad transformers defined there. The *Data.Maybe* module defines useful functions for dealing with optional values of type *Maybe a*, and the module *Data.Map* defines finite maps. These will be used to define environments (variable-value mappings) in our little interpreter.

The following data types for modelling programs in that language will be used:

```
type Name = String                    -- variable names
data Exp   = Lit Integer              -- expressions
           | Var Name
           | Plus Exp Exp
           | Abs Name Exp
           | App Exp Exp
           deriving (Show)
data Value = IntVal Integer           -- values
           | FunVal Env Name Exp
           deriving (Show)
type Env   = Map.Map Name Value       -- mapping from names to values
```

The *Name* type is simply a shorthand for the standard *String* type. It is used to make clear when we are talking about variable names and not about general strings. The *Exp* data type has variants for literal integers (constants), variables, addition, λ expressions (abstractions) and function application. The programs which are to be evaluated will be made up of the *Exp* data type, whereas the results are from the *Value* type. Values are either integers or functions (closures). The *Env* component of a *FunVal* is the environment in which the corresponding λ-abstraction was evaluated.

[3] `http://uebb.cs.tu-berlin.de/~magr/pub/Transformers.lhs`
[4] `http://www.cs.virginia.edu/~wh5a/personal/Transformers.lhs`

Since the example for using monad transformers will be an interpreter for the little language defined above, we start by defining an evaluation function. This function is not monadic and will serve as a kind of "reference implementation". The definition of the interpreter function, called *eval0*, is straightforward.

$$
\begin{aligned}
&\mathit{eval0} && :: \mathit{Env} \to \mathit{Exp} \to \mathit{Value}\\
&\mathit{eval0}\ \mathit{env}\ (\mathit{Lit}\ i) && = \mathit{IntVal}\ i\\
&\mathit{eval0}\ \mathit{env}\ (\mathit{Var}\ n) && = \mathit{fromJust}\ (\mathit{Map.lookup}\ n\ \mathit{env})\\
&\mathit{eval0}\ \mathit{env}\ (\mathit{Plus}\ e1\ e2) && = \textbf{let}\ \mathit{IntVal}\ i1 = \mathit{eval0}\ \mathit{env}\ e1\\
&&&\qquad\quad \mathit{IntVal}\ i2 = \mathit{eval0}\ \mathit{env}\ e2\\
&&&\quad\ \textbf{in}\ \mathit{IntVal}\ (i1 + i2)\\
&\mathit{eval0}\ \mathit{env}\ (\mathit{Abs}\ n\ e) && = \mathit{FunVal}\ \mathit{env}\ n\ e\\
&\mathit{eval0}\ \mathit{env}\ (\mathit{App}\ e1\ e2) && = \textbf{let}\ \mathit{val1} = \mathit{eval0}\ \mathit{env}\ e1\\
&&&\qquad\quad \mathit{val2} = \mathit{eval0}\ \mathit{env}\ e2\\
&&&\quad\ \textbf{in case}\ \mathit{val1}\ \textbf{of}\\
&&&\qquad\quad \mathit{FunVal}\ \mathit{env}'\ n\ \mathit{body} \to \mathit{eval0}\ (\mathit{Map.insert}\ n\ \mathit{val2}\ \mathit{env}')\ \mathit{body}
\end{aligned}
$$

Integer literals simply evaluate to themselves (packaged up in the *Value* data type), variables evaluate to the values to which they are bound in the environment. The use of the *fromJust*[5] function is necessary because the *Map.lookup* function returns a *Maybe* value. Note that the use of this function introduces an error condition: when a variable is used which is not bound anywhere using a λ expression, the program will halt with an error message. Addition is implemented by simply evaluating both operands and returning their sum. Whenever one of the addition operands evaluates to a non-number, the pattern matching in the **let** expression will fail, also terminating the program with an error message. Abstractions simply evaluate to functional values, which capture the environment in which they are evaluated. Function application proceeds similar to addition, by first evaluating the function and the argument. The first expression must evaluate to a functional value, whose body is then evaluated in the captured environment, extended with the binding of the function parameter to the argument value. The **case** expression used here to deconstruct the functional value introduces another error possibility. In later sections of this text, we will handle these error cases using an error monad, which gives us more control over their handling.

The definition of *eval0* could be shortened a little bit, for example, the **let** expression in the *App* case seems superfluous. Nevertheless, the definition given here will make it easier to relate it to the monadic versions defined below.

The following example expression,

$$12 + ((\lambda x \to x)(4 + 2))$$

can be used to test this interpreter and all of the others we will define shortly.

$$\mathit{exampleExp} = \mathit{Lit}\ 12\ `\mathit{Plus}`\ (\mathit{App}\ (\mathit{Abs}\ \texttt{"x"}\ (\mathit{Var}\ \texttt{"x"}))\ (\mathit{Lit}\ 4\ `\mathit{Plus}`\ \mathit{Lit}\ 2))$$

For example, entering

```
eval0 Map.empty exampleExp
```

in `ghci` will give the answer

```
IntVal 18
```

[5] *fromJust* has the type $\mathit{Maybe}\ \alpha \to \alpha$

2 Monad Transformers

The goal of using monad transformers is to have control over aspects of computations, such as state, errors, environments etc. It is a bit tedious to reformulate an already written program in monadic style, but once that is done, it is relatively easy to add, remove or change the monads involved.

In this section, we will first rewrite the example program from Section **??** in monadic style and then extend the data types and function definitions with various monad transformer types and functions step by step.

2.1 Converting to Monadic Style

In order to use monad transformers, it is necessary to express functions in monadic style. That means that the programmer needs to impose sequencing on all monadic operations using **do** notation, and to use the *return* function in order to specify the result of a function.

First, we define a monad in which the evaluator will be defined. The following type synonym defines *Eval1* α as a synonym for the type *Identity* α. *Identity* is a monad imported from *Control.Monad.Identity*, which is perhaps the simplest monad imaginable: it defines the standard *return* and $\gg=$ operations for constructing operations in the monad, and additionally a function *runIdentity* to execute such operations. Other than that, the identity monad has no effect. In some sense, we will use this monad as a "base case", around which other monad transformers can be wrapped. For readability, we also define a function *runEval1*, which simply calls *runIdentity*.

```
type Eval1 α = Identity α

runEval1       :: Eval1 α → α
runEval1 ev    = runIdentity ev
```

Based on the *Eval1* monad, we now rewrite the *eval0* function as *eval1*:

```
eval1                    :: Env → Exp → Eval1 Value
eval1 env (Lit i)        = return $ IntVal i
eval1 env (Var n)        = Map.lookup n env
eval1 env (Plus e1 e2) = do IntVal i1 ← eval1 env e1
                            IntVal i2 ← eval1 env e2
                            return $ IntVal (i1 + i2)
eval1 env (Abs n e)      = return $ FunVal env n e
eval1 env (App e1 e2) = do val1 ← eval1 env e1
                           val2 ← eval1 env e2
                           case val1 of
                             FunVal env' n body →
                               eval1 (Map.insert n val2 env') body
```

The first thing to note is that the cases for *Lit* and *Abs* use the *return* function for specifying their result.[6] The next is that the *Var* case does not need a *fromJust* call anymore: The reason is that *Map.lookup* is defined to work within any monad by simply calling the monad's *fail* function – this fits nicely with our monadic formulation here. (The *fail* function of the *Maybe* monad returns *Nothing*, whereas the *fail* function in the *Identity* monad throws an exception, which will lead to different error messages.)

The *Plus* and *App* cases now evaluate their subexpressions using **do**-notation, binding their results to variables. In the *Plus* case, the result is returned using *return*, whereas in th *App* case, the function value is further discriminated like in the *eval0* function above.

[6] The ($) operator is function application with low precedence and mainly used to avoid parentheses.

In order to test this interpreter, we have to evaluate the monadic action obtained by applying *eval1* to our example expression *exampleExp*. This is done by calling *runEval1* defined earlier:

```
runEval1 (eval1 Map.empty exampleExp)
```

gives

```
IntVal 18
```

To recapitulate: conversion to monadic form consists mainly of returning function results using the *return* function, and sequencing of monadic actions using **do** notation or the $\gg=$ or $\gg$ (monadic bind) functions.

Note: The type of *eval1* could be generalized to

$$eval1 :: Monad\ m \Rightarrow Env \rightarrow Exp \rightarrow m\ Value,$$

because we do not use any monadic operations other than *return* and $\gg=$ (hidden in the **do** notation). This allows the use of *eval1* in any monadic context, so that instead of

```
runEval1 (eval1 Map.empty exampleExp)
```

we could write

```
eval1 Map.empty exampleExp
```

at the `ghci` prompt. This would run the expression in the *IO* monad, because internally the interpreter uses the *print* function, which lives in just this monad.[7] In some contexts, this is a nice feature, but in general you will be using some operations specific to a particular monad, and this forces your operation to stay within that special monad.

2.2 A Few Words on Monads

We need to clarify some concept before moving on to monads and monad transformers. A data type T is a monad if it is an instance of the *Monad* class. That is, we provided two functions *return* and ($\gg=$) for T, whose implementation depends on the very semantics of T. Besides the constraints on the two functions' types, they must obey the monad laws too, but this constraint is not decidable by algorithms, and thus has to be enforced by programmers themselves.

The monad class is only an interface that specifies what a type T must provide to become a monad. The real strengths of monads come from the power of type classes supported by the strong type system, and the useful monad types predefined in the monad transformer library (*mtl*). For instance, the *Either* type is a monad:

```
data  Either a b  =  Left a | Right b   deriving (Eq, Ord )

instance (Error e) => Monad (Either e) where
    return        = Right
    Left  l >>= _ = Left l
    Right r >>= k = k r
    fail msg      = Left (strMsg msg)
```

The *Monad* class is further extended to more useful classes, including *MonadError*, *MonadReader*, *MonadWriter*, etc. These extended classes can be thought of as refined interfaces that require more functions to be implemented by their instances. For example:

[7] This only works in recent versions of GHC and unfortunately not in Hugs.

```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a

class (Monad m) => MonadReader r m | m -> r where
    ask   :: m r
    local :: (r -> r) -> m a -> m a
```

Although programmers can instantiate their own data types by defining the required functions, it is always nice to make use of existing code:

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
    return a = Reader $ \_ -> a
    m >>= k  = Reader $ \r -> runReader (k (runReader m r)) r

instance MonadReader r (Reader r) where
    ask       = Reader id
    local f m = Reader $ runReader m . f

newtype Writer w a = Writer { runWriter :: (a, w) }

instance (Monoid w) => Monad (Writer w) where
    return a = Writer (a, mempty)
    m >>= k  = Writer $ let
        (a, w)  = runWriter m
        (b, w') = runWriter (k a)
        in (b, w `mappend` w')

instance (Monoid w) => MonadWriter w (Writer w) where
    tell   w = Writer ((), w)
    listen m = Writer $ let (a, w) = runWriter m in ((a, w), w)
    pass   m = Writer $ let ((a, f), w) = runWriter m in (a, f w)

instance (Error e) => MonadError e (Either e) where
    throwError             = Left
    Left  l `catchError` h = h l
    Right r `catchError` _ = Right r
```

In general, a type X (Cont, Reader, Writer, State, etc.) is an instance of *Monad*, and an instance of *MonadX* as well. But *Error* is an exception here! *Error* is a class, not a type, and not a subclass of *Monad*, so do not get confused.

What if we want to add error handling, state, logging, and I/O together? Most likely we do not want to reinvent the wheel and instantiate our data type for each of those classes. This is how monad transformers come into play! Basically, we use a stack of monad transformers to morph our original type, with the innermost monad being *Identity* or *IO*. We will explain the details below.

2.3 Adding Error Handling

We have already seen that our evaluation function is partial, that means it will terminate with an error message for some inputs, for example for expressions with unbound variables or type

errors.

Using monad transformers, we simply go to our local monad transformer library and take the *ErrorT* monad transformer, using it to extend the basic *Eval1* monad to *Eval2*.

```
type Eval2 α = ErrorT String Identity α
```

The *String* type argument to *ErrorT* is the type of exceptions, that is the values which are used to indicate error conditions. We use *String* here to keep things simple, in a real implementation we might want to include source code locations (in a compiler) or time stamps (in some kind of web application).

The function for running a computation in the *Eval2* monad changes in two ways. First, the result of evaluation is now of type *Either String* α, where the result *Left s* indicates that an error has occurred with error message s, or *Right r*, which stands for successful evaluation with result r. Second, we need to call the function *runErrorT* on the given computation to yield an *Identity* computation, which can in turn be evaluated using *runIdentity*.

```
runEval2     :: Eval2 α → Either String α
runEval2 ev = runIdentity (runErrorT ev)
```

We can now simply change the type of our *eval1* function, giving the following version, called *eval2a*.

```
eval2a                    :: Env → Exp → Eval2 Value
eval2a env (Lit i)        = return $ IntVal i
eval2a env (Var n)        = Map.lookup n env
eval2a env (Plus e1 e2) = do IntVal i1 ← eval2a env e1
                             IntVal i2 ← eval2a env e2
                             return $ IntVal (i1 + i2)
eval2a env (Abs n e)      = return $ FunVal env n e
eval2a env (App e1 e2) = do val1 ← eval2a env e1
                            val2 ← eval2a env e2
                            case val1 of
                              FunVal env' n body →
                                eval2a (Map.insert n val2 env') body
```

This version can be run using the *runEval2* function defined above. When we apply this function to our example expression, the result only varies in that it is wrapped in a *Right* constructor:

```
runEval2 (eval2a Map.empty exampleExp) ⇒ Right (IntVal 18)
```

But unfortunately, when given an invalid expression, the error reporting of the *ErrorT* transformer is not used. We have to modify our definition in order to give useful error messages:

```
eval2b                    :: Env → Exp → Eval2 Value
eval2b env (Lit i)        = return $ IntVal i
eval2b env (Var n)        = Map.lookup n env
eval2b env (Plus e1 e2) = do e1' ← eval2b env e1
                             e2' ← eval2b env e2
                             case (e1', e2') of
                               (IntVal i1, IntVal i2) →
                                 return $ IntVal (i1 + i2)
                               _ → throwError "type error"
eval2b env (Abs n e)      = return $ FunVal env n e
eval2b env (App e1 e2) = do val1 ← eval2b env e1
```

```
                    val2 ← eval2b env e2
                    case val1 of
                      FunVal env' n body →
                        eval2b (Map.insert n val2 env') body
                      _ → throwError "type error"
```

Now, when we try to evaluate an invalid expression, we get an error message, packaged in the *Left* constructor. So by matching against the result of an evaluation, we can distinguish between normal and error results.

```
runEval2 (eval2a Map.empty (Plus (Lit 1) (Abs "x" (Var "x")))) ⇒
  Left "type error"
```

But wait a minute! What is about *Map.lookup n env*? Shouldn't we check whether it returns *Nothing* and generate an appropriate error message? As mentioned above, *Map.lookup* returns its result in an arbitrary monad, and the *Control.Monad.Error* module gives the necessary definitions so that it works just out of the box:

```
runEval2 (eval2b Map.empty (Var "x")) ⇒
  Left "Data.Map.lookup: Key not found"
```

A little bit of closer inspection of function *eval2b* reveals that we can do even shorter (better?) by exploiting the fact that monadic binding in a **do** expression uses the *fail* function whenever a pattern match fails. And, as we have seen, the *fail* function does what we want.

```
eval2c                  :: Env → Exp → Eval2 Value
eval2c env (Lit i)      = return $ IntVal i
eval2c env (Var n)      = Map.lookup n env
eval2c env (Plus e1 e2) = do IntVal i1 ← eval2c env e1
                             IntVal i2 ← eval2c env e2
                             return $ IntVal (i1 + i2)
eval2c env (Abs n e)    = return $ FunVal env n e
eval2c env (App e1 e2)  = do FunVal env' n body ← eval2c env e1
                             val2               ← eval2c env e2
                             eval2c (Map.insert n val2 env') body
```

The drawback of this function is that the error messages only talks about "pattern match failure", with no specific information about why the pattern match fails. Thus, in order to get good error messages, it is better to provide our own calls to *throwError*. This is what we'll do for the final version of the error handling evaluation.

```
eval2                   :: Env → Exp → Eval2 Value
eval2 env (Lit i)       = return $ IntVal i
eval2 env (Var n)       = case Map.lookup n env of
                            Nothing → throwError ("unbound variable: " ++ n)
                            Just val → return val
eval2 env (Plus e1 e2)  = do e1' ← eval2 env e1
                             e2' ← eval2 env e2
                             case (e1', e2') of
                               (IntVal i1, IntVal i2) →
                                 return $ IntVal (i1 + i2)
                               _ → throwError "type error in addition"
eval2 env (Abs n e)     = return $ FunVal env n e
eval2 env (App e1 e2)   = do val1 ← eval2 env e1
```

```
                val2 ← eval2 env e2
                case val1 of
                  FunVal env' n body →
                    eval2 (Map.insert n val2 env') body
                  _ → throwError "type error in application"
```

Note: The *Control.Monad.Error* module provides another function for catching errors raised using *throwError*, called *catchError* :: $m\ a \to (e \to m\ a) \to m\ a$ for arbitrary error monads. It can be used for either handling errors locally or passing them on to the surrounding calling context.

2.4 Hiding the Environment

One way to make the definition of the evaluation function even more pleasing is to hide the environment from all function definitions and calls. Since there is only one place where the environment is extended (for function application) and two places where it is actually used (for variables and λ expressions), we can reduce the amount of code by hiding it in all other places. This will be done by adding a *ReaderT* monad transformer in order to implement a reader monad. A reader monad passes a value into a computation and all its sub-computations. This value can be read by all enclosed computations and get modified for nested computations. In contrast to state monads (which will be introduced in Section **??**), an encapsulated computation cannot change the value used by surrounding computations.

We start by simply wrapping a *ReaderT* constructor around our previous monad.

```
type Eval3 α = ReaderT Env (ErrorT String Identity) α
```

The run function *runEval3* must be slightly modified, because we need to pass in the initial environment. The reason is that we will remove the environment parameter from the evaluation function.

```
runEval3 :: Env → Eval3 α → Either String α
runEval3 env ev = runIdentity (runErrorT (runReaderT ev env))


eval3                :: Exp → Eval3 Value
eval3 (Lit i)        = return $ IntVal i
eval3 (Var n)        = do env ← ask
                          case Map.lookup n env of
                            Nothing → throwError ("unbound variable: " ++ n)
                            Just val → return val
eval3 (Plus e1 e2) = do e1' ← eval3 e1
                        e2' ← eval3 e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) →
                            return $ IntVal (i1 + i2)
                          _ → throwError "type error in addition"
eval3 (Abs n e)      = do env ← ask
                          return $ FunVal env n e
eval3 (App e1 e2)  = do val1 ← eval3 e1
                        val2 ← eval3 e2
                        case val1 of
                          FunVal env' n body →
```

```
                local (const (Map.insert n val2 env'))
                  (eval3 body)
            _ → throwError "type error in application"
```

For our running example, we now have to evaluate

```
runEval3 Map.empty (eval3 exampleExp)
```

In all places where the current environment is needed, it is extracted from the hidden state of the reader monad using the *ask* function. In the case of function application, the *local* function is used for modifying the environment for the recursive call. Local has the type $(r \to r) \to m\ a \to m\ a$, that is we need to pass in a function which maps the current environment to the one to be used in the nested computation, which is the second argument. In our case, the nested environment does not depend on the current environment, so we simply pass in a constant function using *const*.

Note: In addition to *ask*, a function *asks* is predefined, which expects a function mapping the environment to a value. This can be used to extract individual components of the environment by applying *asks* to record selector functions.

2.5 A Few Words on Monad Transformers

As we have mentioned, *ErrorT* transforms a monad into a *MonadError*:

```
newtype ErrorT e m a = ErrorT { runErrorT :: m (Either e a) }

instance (Monad m, Error e) => Monad (ErrorT e m) where
    return a = ErrorT $ return (Right a)
    m >>= k  = ErrorT $ do
        a <- runErrorT m
        case a of
            Left  l -> return (Left l)
            Right r -> runErrorT (k r)
    fail msg = ErrorT $ return (Left (strMsg msg))

instance (Monad m, Error e) => MonadError e (ErrorT e m) where
    throwError l     = ErrorT $ return (Left l)
    m `catchError` h = ErrorT $ do
        a <- runErrorT m
        case a of
            Left  l -> runErrorT (h l)
            Right r -> return (Right r)
```

After we add another layer of *ReaderT* on the top, we get a *MonadReader*. So we are able to call *ask* and *local* in *eval3*. But we also need to call functions provided by the monads buried inside ReaderT too! In this case, we want to call *throwError* provided by MonadError. We could abstract this pattern with a new class:

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

The *lift* function is supposed to be composed with functions whose return type corresponds to the inner monad. In some sense, *lift* lifts the return value of a function up by one layer in

the monad stack. Or, to think in a more intuitive way, *lift* sends your command inwards by one layer. To access a function *foo* provided by a monad three layers down the stack, you need to compose lift three times: `lift $ lift $ lift $ foo`.

Back to our example. As we should be expecting, the two transformers ReaderT and ErrorT really are instances of the MonadTrans class:

```
instance MonadTrans (ReaderT r) where
    lift m = ReaderT $ \_ -> m

instance (Error e) => MonadTrans (ErrorT e) where
    lift m = ErrorT $ do
        a <- m
        return (Right a)
```

At this point you should be wondering, why were we able to call *throwError* in *eval3* without first lifting it? The answer is because the mtl writers decided to save us some time by instantiating ReaderT as a MonadError. In fact, an ErrorT is a MonadReader too:

```
instance (MonadError e m) => MonadError e (ReaderT r m) where
    throwError       = lift . throwError
    m `catchError` h = ReaderT $ \r -> runReaderT m r
        `catchError` \e -> runReaderT (h e) r

instance (Error e, MonadReader r m) => MonadReader r (ErrorT e m) where
    ask       = lift ask
    local f m = ErrorT $ local f (runErrorT m)
```

The mtl writers even went through all the trouble and made the monad transformers instances of each other (that is n^2 instances)! If you need to build a new monad transformer yourself, think carefully about the design of all the plumbing behind the scene.

Down in this document, we call *liftIO* in *eval6* to perform I/O actions. Why do we need to lift in this case? Because there is no IO class for which we can instantiate a type as. Therefore, for I/O actions, we have to call lift to send the commands inwards. For *eval6*, we would need to compose *lift* four times to print something. This is just inconvenient, so people create a new class *MonadIO* such that we only need to call *liftIO* once, without having to keep count of how many times to compose *lift*:

```
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO = id

instance (Error e, MonadIO m) => MonadIO (ErrorT e m) where
    liftIO = lift . liftIO

instance (MonadIO m) => MonadIO (ReaderT r m) where
    liftIO = lift . liftIO
```

Finally, let us study the types from *runEval1* to *runEval6* intuitively. *runEval1* is easy. For the other functions, they pick up the right types and compose them together along the way as they peel the onion of monads. Just pay attention to how their types change as we extend these functions. For example, let us decide the return value for *runEval4*. Ignoring

ReaderT as it does not affect the return value (although it does affect runEval4's arguments), runEval4 first peels off ErrorT and constructs a value of type *Either String a*. Next, it peels off StateT and constructs a pair whose first component is the value being computed, and whose second component is the side effect, i.e., the state. Therefore, the type of the final result is $(Either\ String\ a, Integer)$. In contrast, *runEval4'* first peels off StateT and then ErrorT. Hence we get $Either\ String\ (a, Integer)$.

To learn more about monads and monad transformers in practice, read All About Monads, Monad transformers on WikiBooks, Monad on haskell.org, and Write Yourself a Scheme in 48 Hours. For more category theory stuff, start with Category theory on WikiBooks, The Haskell Programmer's Guide to the IO Monad — Don't Panic, and Monads for Programming Languages.

2.6 Adding State

Another important application of monads is to provide mutable state to otherwise purely functional code. This can be done using a *State* monad, which provides operations for specifying an initial state, querying the current state and changing it.

As an example, suppose that we want to add profiling capabilities to our little interpreter. We define the new monad by wrapping a *StateT* constructor around the innermost monad, *Identity*. (In the case of *State* and *Error* monads, the order of these constructor matters, as we will see below.) The state maintained in our example is a simple integer value, but it could be a value of any data type we wish. Normally, it will be a record holding the complete state necessary for the task at hand.

```
type Eval4 α = ReaderT Env (ErrorT String (StateT Integer Identity)) α
```

The return type of the function *runEval4* changes, because the final state is returned together with the evaluation result (error or value). Additionally, we give the initial state as an additional parameter so that we gain some flexibility (this can be used, for example, to start a computation in the final state of another one).

```
runEval4 :: Env → Integer → Eval4 α → (Either String α, Integer)
runEval4 env st ev = runIdentity (runStateT (runErrorT (runReaderT ev env)) st)
```

For our simple example, we only want to count the number of evaluation steps, that is the number of calls to the *eval4* function. All modification happens in a little helper function *tick*, which gets the hidden state from the computation, increases the counter and stores it back. The type of *tick* is not $Eval4\ ()$, as should be expected, because we plan to re-use it in other sections below. Therefore, we simply state that the monad in which *tick* will be used must be a state monad, and that the state manipulated in that monad is numeric, so that we can use the (+) operator on it.

```
tick :: (Num s, MonadState s m) ⇒ m ()
tick = do st ← get
          put (st + 1)
```

By adding a call to the *tick* function in each case, we can count the number of applications.

```
  -- eval4 :: Exp -¿ Eval4 Value
eval4 (Lit i)      = do tick
                        return $ IntVal i
eval4 (Var n)      = do tick
                        env ← ask
                        case Map.lookup n env of
                          Nothing → throwError ("unbound variable: " ++ n)
```

```
                              Just val → return val
eval4 (Plus e1 e2) = do tick
                        e1' ← eval4 e1
                        e2' ← eval4 e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) →
                            return $ IntVal (i1 + i2)
                          _ → throwError "type error in addition"
eval4 (Abs n e)    = do tick
                        env ← ask
                        return $ FunVal env n e
eval4 (App e1 e2) = do tick
                        val1 ← eval4 e1
                        val2 ← eval4 e2
                        case val1 of
                          FunVal env' n body →
                            local (const (Map.insert n val2 env'))
                              (eval4 body)
                          _ → throwError "type error in application"
```

Evaluating our example expression yields:

```
(Right (IntVal 18),8)
```

meaning that the evaluation was successful, returned the integer 18 and took 8 reduction steps.

Note: When the type of the *Eval4* monad is changed to the following (*StateT* and *ErrorT* are swapped), the interpretation of the monad changes.

```
type Eval4' α = ReaderT Env (StateT Integer (ErrorT String Identity)) α
```

Instead of returning a result (error or normal) and a state, either an error or a result together with the final state is returned, as can be seen in the type of the corresponding run function:

```
runEval4' :: Env → Integer → Eval4' α → (Either String (α, Integer))
runEval4' env st ev = runIdentity (runErrorT (runStateT (runReaderT ev env) st))
```

The position of the reader monad transformer does not matter, since it does not contribute to the final result.

Note: The *State* monad also provides an additional function, *gets* which applies a projection function to the state before returning it. There is also a function *modify* which can be used to change the internal state by applying a function over it.

2.7 Adding Logging

The last monad transformer in the toolbox which will be described here is *WriterT*. It is in some sense dual to *ReaderT*, because the functions it provides let you add values to the result of the computation instead of using some values passed in.

```
type Eval5 α = ReaderT Env (ErrorT String
                             (WriterT [String] (StateT Integer Identity))) α
```

Similar to *StateT*, *WriterT* interacts with *ErrorT* because it produces output. So depending on the order of *ErrorT* and *WriterT*, the result will include the values written out or not when

an error occurs. The values to be written out will be lists of strings. When you read the documentation for the *WriterT* monad transformer, you will notice that the type of the output values is restricted to be a member of the type class *Monoid*. This is necessary because the methods of this class are used internally to construct the initial value and to combine several values written out.

The running function is extended in the same way as earlier.

```
runEval5           :: Env → Integer → Eval5 α → ((Either String α, [String]), Integer)
runEval5 env st ev =
  runIdentity (runStateT (runWriterT (runErrorT (runReaderT ev env))) st)
```

In the evaluation function, we illustrate the use of the writer monad by writing out the name of each variable encountered during evaluation.

```
eval5              :: Exp → Eval5 Value
eval5 (Lit i)      = do tick
                        return $ IntVal i
eval5 (Var n)      = do tick
                        tell [n]
                        env ← ask
                        case Map.lookup n env of
                          Nothing → throwError ("unbound variable: " ++ n)
                          Just val → return val
eval5 (Plus e1 e2) = do tick
                        e1' ← eval5 e1
                        e2' ← eval5 e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) →
                            return $ IntVal (i1 + i2)
                          _ → throwError "type error in addition"
eval5 (Abs n e)    = do tick
                        env ← ask
                        return $ FunVal env n e
eval5 (App e1 e2)  = do tick
                        val1 ← eval5 e1
                        val2 ← eval5 e2
                        case val1 of
                          FunVal env' n body →
                            local (const (Map.insert n val2 env'))
                              (eval5 body)
                          _ → throwError "type error in application"
```

2.8 What about I/O?

Until now, we have not considered one importand aspect: input and output. How do we integrate I/O into the monadic definitions we have developed so far? It is not possible to define an I/O monad transformer, because the execution of I/O operations in Haskell cannot be arbitrarily nested into other functions or monads, they are only allowed in the monad *IO*. Fortunately, the monad transformer library provides us with the infrastructure to easily integrate I/O operations into our framework: we simply substitute *IO* where we have used *Identity*! This is possible because *Identity* is the base monad, and as we have seen, the function *runIdentity* for evaluating actions in this monad is always applied last.

```
type Eval6 α = ReaderT Env (ErrorT String
                              (WriterT [String] (StateT Integer IO))) α
```

The return type of *runEval6* is wrapped in an *IO* constructor, which means that the running an *Eval6* computation does not directly yield a result, but an I/O computation which must be run in order to get at the result. Accordingly, the *runIdentity* invocation disappears.

```
runEval6 :: Env → Integer → Eval6 α → IO ((Either String α, [String]), Integer)
runEval6 env st ev =
  runStateT (runWriterT (runErrorT (runReaderT ev env))) st
```

In the *eval6* function we can now use I/O operations, with one minor notational inconvenience: we have to invoke the operations using the function *liftIO*, which lifts the I/O computation into the currently running monad. As an example, we chose to print out each integer constant as soon as it is evaluated. (We don't think this is good style, but it illustrates the point and sometimes makes a good debugging technique.)

```
eval6                :: Exp → Eval6 Value
eval6 (Lit i)        = do tick
                          liftIO $ print i
                          return $ IntVal i
eval6 (Var n)        = do tick
                          tell [n]
                          env ← ask
                          case Map.lookup n env of
                            Nothing → throwError ("unbound variable: " ++ n)
                            Just val → return val
eval6 (Plus e1 e2) = do tick
                          e1' ← eval6 e1
                          e2' ← eval6 e2
                          case (e1', e2') of
                            (IntVal i1, IntVal i2) →
                              return $ IntVal (i1 + i2)
                            _ → throwError "type error in addition"
eval6 (Abs n e)      = do tick
                          env ← ask
                          return $ FunVal env n e
eval6 (App e1 e2) = do tick
                          val1 ← eval6 e1
                          val2 ← eval6 e2
                          case val1 of
                            FunVal env' n body →
                              local (const (Map.insert n val2 env'))
                                (eval6 body)
                            _ → throwError "type error in application"
```

3 Conclusion

Monad transformers are a powerful tool in the toolbox of a functional programmer. This paper introduces several of the monad transformers available in current Haskell implementations, and shows how to use and combine them in the context of a simple functional interpreter.

We have not covered all monad transformers presently implemented in Haskell (e.g., continuation and list monad transformers) and recommend to read the library documentation available from the Haskell web site for additional information.

The use of monad transformers makes it very easy to define specialized monads for many applications, reducing the temptation to put everything possibly needed into the one and only monad hand-made for the current application.

Happy hacking in Haskell![8]

Acknowledgements

Thanks to Christian Maeder, Bruno Martínez and Tomasz Zielonka for their valuable feedback and suggestions for improvement.

[8]The main function is defined at the end of the source file, not shown in this PDF document.