Adventures in Three Monads

by Edward Z. Yang ⟨ezyang@mit.edu⟩

January 11, 2010

The standard monad libraries define a number of "bread and butter" monads, including the State, Reader, Writer, List and Maybe monads. However, they are not the only monads available to an enterprising Haskeller's toolbox. In this text, we look at three other monads—the Logic monad, the Prompt monad, and the Failure monad—each of which tackle a common problem found in the engineering of programs.

The Logic monad

The Logic monad [1] implements "backtracking computations." It is provided by the `logict` package. Backtracking is a general approach that can be applied to many search problems: given a partial solution, we determine if we can generate any further partial solutions; if we cannot, we discard this partial solution and backtrack to an earlier point in the search tree. Backtracking tends to be faster than brute-force enumeration of the solution space, because if a partial solution violates a constraint, then further extensions of that partial solution can be eliminated, a process called pruning.

Strictly speaking, we don't need the Logic monad to implement backtracking: the List monad gives us nondeterministic computation, and over finite search spaces both the Logic monad and the List monad can give us the same answers. However, the Logic monad is much more efficient due to an underlying continuation-based implementation. Additionally, the `MonadLogic` typeclass exposes a few more operators that allow us to control when to perform a computation; this is a common practice in monad libraries, since it lets the interface be divorced from the actual implementation, whether it is the Logic monad, the LogicT transformer, or even the List monad. In this section, we will explicitly use `Logic` for brevity.

List monad equivalence

The Logic monad implements a strict superset of the List monad; as such, anything in the List monad can be directly translated into the Logic monad. Function signatures that have a type `[a]` now have a type `Logic a`; data in the List monad is transformed accordingly:

`[1]`	$\Rightarrow$	`return 1`
`[]`	$\Rightarrow$	`mzero`
`++ :: [a] -> [a] -> [a]`	$\Rightarrow$	`mplus :: m a -> m a -> m a`
`concat :: [[a]] -> [a]`	$\Rightarrow$	`msum :: [m a] -> m a`
`[1,2,3]`	$\Rightarrow$	`(msum . map return) [1,2,3]`

Notice that many of these transformations are simply generalizations of lists into the MonadPlus context. If we explicitly change `m a` to `[a]` in the type signatures of `mplus` and `msum`, we get back the original list operations.

To get data back out of the Logic monad (and back into lists), you can use the following transformations:

`id :: [a] -> [a]`	$\Rightarrow$	`observeAll :: Logic a -> [a]`
`take :: Int -> [a] -> [a]`	$\Rightarrow$	`observeMany :: Int -> Logic a -> [a]`
`head :: [a] -> a`	$\Rightarrow$	`observe :: Logic a -> a`

You can see a brief example this equivalence in Listing 1.

```
choices :: MonadPlus m => [a] -> m a
choices = msum . map return

evensList :: [Int]
evensList = do
    n <- [1..]
    if n `mod` 2 == 0
        then [n]
```

```
        else []

evensLogic :: Logic Int
evensLogic = do
    n <- choices [1..]
    if n `mod` 2 == 0
        then return n
        else mzero
evensList' = observeAll evensLogic
```

Listing 1: Code in the List and Logic monads

Nondeterminism and backtracking

In `do` notation, the `<-` operator extracts a pure value from the monad. In the Logic monad, as in the List monad, the successive code takes on each value from the list in turn, and the results get concatenated together, similar to a `fork` operation in Unix. However, viewed as a search, the `<-` represents a branching operation: the list represents possible extensions of the current candidate solution, and we now select a single extension to further conduct search on.

The power in any monad is to hide away "incidental" details; in the case of the Logic monad the incidental detail is that we're doing a search over a nondeterministic computation, and write code as if it were deterministic. It's a little difficult to see this in a toy example, so instead we will develop code for a nondeterministic Turing machine.

A Turing machine consists of a tape, which we represent as two infinite lists of symbols, a head, which can write to a single symbol on the Turing machine, an action table, which we represent as an array and a state register. An implementation of these data structures is shown in Listing 2. We also have two basic functions for manipulating the tape—writing and moving—in Listing 3. Our particular implementation is a two-state, five-symbol Turing machine.

A Turing machine operates by using its current state and symbol underneath the head to index into the action table. A deterministic Turing machine would have a single transition which encodes what the next state is, what symbol should be written to the tape, and what direction the head should move after writing the symbol. A nondeterministic Turing machine would have many possible transitions for each index, and the operator would be expected to keep track of the branching—thus `DTuringTransition` encodes a deterministic transition, whereas `TuringTransition` is nondeterministic.

We omit the deterministic implementation of a single step of running the Turing machine, precisely because the nondeterministic implementation in Listing 4 still communicates the essence of Turing machine execution clearly and can easily simulate the deterministic version.

If we'd like to use our nondeterministic Turing machine to search the space of deterministic Turing machines, we need to slightly modify the behavior of `stepMachine`: specifically, any time we make a choice with `<-`, we should stick with that choice for the rest of the machine's execution (notice, in the original implementation, that `machine` is a return value but is unchanged!) Amazingly, Listing 5 requires only a single extra line of book-keeping (marked by `-- *`).

From there, performing a search for a machine involves having an initial "every machine" Turing machine, implemented in Listing 6 by filling every entry in the action table with "every transition", and then stepping through it and pruning results that don't halt or that give the wrong answer.

```haskell
data RunningTuringMachine = RTM
    { rtmMachine :: TuringMachine
    , rtmTape    :: Tape
    , rtmState   :: TuringState
    } deriving (Show)

type TuringMachine      = Array TuringIndex TuringTransition
type TuringIndex        = (State, Symbol)
type TuringAction       = (TuringIndex, TuringTransition)
type TuringTransition   = Logic DTuringTransition
type DTuringTransition  = (TuringState, Symbol, TuringMove)

data Tape               = Tape [Symbol] Symbol [Symbol]
data TuringState        = Halt | State State
    deriving (Eq, Ord, Show)
data State              = StateA | StateB
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
data TuringMove         = MoveRight | MoveLeft | Stay
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
data Symbol             = SB | S0 | S1 | SL | SR
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
```

Listing 2: Data types for a nondeterministic Turing machine

```
moveTape :: TuringMove -> Tape -> Tape
moveTape Stay x = x
moveTape MoveRight (Tape left cur right) =
                   Tape (tail left)  (head left) (cur :right)
moveTape MoveLeft  (Tape left cur right) =
                   Tape (cur :left) (head right) (tail right)

writeTape :: Symbol -> Tape -> Tape
writeTape s (Tape left _ right) = Tape left s right
```

Listing 3: Tape manipulation functions

```
stepMachine :: RunningTuringMachine -> Logic RunningTuringMachine
stepMachine rtm@(RTM _ _ Halt) = return rtm
stepMachine (RTM machine tape@(Tape _ cur _) (State state)) = do
    (state’, cur’, move) <- machine ! (state, cur)
    let tape’ = moveTape move $ writeTape cur’ tape
    return $ RTM machine tape’ state’
```

Listing 4: Nondeterministic step

```
stepMachine’ :: RunningTuringMachine -> Logic RunningTuringMachine
stepMachine’ rtm@(RTM _ _ Halt) = return rtm
stepMachine’ (RTM machine tape@(Tape _ cur _) (State state)) = do
    trans@(state’, cur’, move) <- machine ! (state, cur)
    let tape’ = moveTape move $ writeTape cur’ tape
        machine’ = machine // [((state, cur), return trans)] -- *
    return $ RTM machine’ tape’ state’
```

Listing 5: Step that collapses nondeterministic

```haskell
-- Generates all values of a bounded, indexable data type.
generate :: (Ix a, Bounded a) => [a]
generate = range (minBound, maxBound)

everyTransition :: TuringTransition
everyTransition = msum . map return $ generate

everyMachine :: TuringMachine
everyMachine = array (minBound, maxBound) $
                      zip (range (minBound, maxBound))
                          (repeat everyTransition)
```

Listing 6: Every transition, every machine

Fair disjunctions

A disjunction occurs whenever you combine the results of the Logic monad with `mplus`. The new monad `a ‘mplus‘ b` will return the results of `a` first and the results of `b` second. Shown in Listing 7 is a simple use of `mplus` to express all integers.

```
naiveIntegers :: Logic Integer
naiveIntegers = return 0 ‘mplus‘
    choices [1..] ‘mplus‘ choices [-1,-2..]
```

Listing 7: Naive representation of $\mathbb{Z}$

Unfortunately, if we try actually observing results from `integers` we find that it never returns any negative numbers: `choices [1..]` succeeds an infinite number of times.

The Logic monad exposes an alternative `mplus` called `interleave`, which interleaves the results of the monads it is combining, so that Listing 8 returns the following sequence:

$$0, 1, -1, 2, -2, 3, -3, 4, -4, \cdots$$

which guarantees that any integer n will be seen in finite time.

```
fairIntegers :: Logic Integer
fairIntegers = return 0 `mplus`
    (choices [1..] `interleave` choices [-1,-2..])
```

Listing 8: Representation of $\mathbb{Z}$ with fair disjunctions

If there are more than two monads to interleave, care must be taken: the `interleave` operator is no longer associative. For example,

```
a `interleave` (b `interleave` c)
```

favors results of `a`, whereas,

```
(a `interleave` b) `interleave` c
```

favors results of `c`. If we can make a balanced full binary tree, we can be completely fair:

```
(a `interleave` c) `interleave` (b `interleave` d)
```

returns:

$$a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, \cdots$$

Fair conjunctions

We will unite the white rose and the red:—
Smile heaven upon this fair conjunction,
That long have frown'd upon their emnity!
—William Shakespeare, Richard III

A conjunction occurs whenever you extract a variable from the logic monad with `>>=`; it is associated with choice. Shown in Listing 9 is a simple search for factorizations of an integer over $2^{\mathbb{N}} \times \mathbb{N}$.

```
nat :: Logic Int
nat = choices [0..]

naiveFactorize :: Int -> Logic (Int, Int)
naiveFactorize n =
    nat >>= \x ->
```

```
  nat >>= \y ->
  guard (2^x * y == n) >>
  return (x, y)
```

Listing 9: Naive factorization

Unfortunately, this code will only ever find a single pair of factors for any given number: $(2^0, n)$. If we remove the `guard` line, we discover why: because the naturals are infinite, the monad has gotten "stuck" on $x = 0$, and we never try $x = \{1, 2, 3, \cdots\}$.

The distributivity law for `MonadPlus`:

```
(mplus a b) >>= k = mplus (a >>= k) (b >>= k)
```

suggests that we can use `interleave` to implement fair conjunctions. Logic does so, and exposes an alternate bind `>>-`. Instead of pursuing a computation of a single choice to completion, it pursues the computation until a single result is found, and then begins computation of a different choice. This has some subtle implications, as can be seen in the restructuring of factorize in Listing 10.

We have split factorization into a generation step (which utilizes fair conjunctions) and a filtering step. Without this separation, the function diverges after returning two results. The Logic monad schedules computations for each of its choices. The fair conjunction is not actually completely “fair”: if it were, it would


```
fairGenerate :: Logic (Int, Int)
fairGenerate =
    nat >>- \x ->
    nat >>- \y ->
    return (x, y)

fairFactorize :: Int -> Logic (Int, Int)
fairFactorize n =
    fairGenerate >>= \(x, y) ->
    guard (2^x * y == n) >>
    return (x, y)
```

Listing 10: Factorization with fair conjunctions never let any choice return more than one result in the case of infinitely many choices, and execution would look like:

$$0, 1, 2, 3, 4, \cdots$$

Instead, the Logic monad has a binary fair conjunction $\wedge$ and applies it recursively to the (unbalanced) tree of choices $c_1 \wedge (c_2 \wedge (c_3 \wedge \ldots))$, asymptotically allocating $1/2^k$ of processing for the kth choice. Execution in this case, assuming that each choice requires infinite computation, looks like:

$$0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, \cdots$$

Recall that the Logic monad executes the first choice $x = 0$ and only switches to the next choice once a result is observed. The non-termination is then the writing on the wall: factorize kicks out results for $x = 0$ and $x = 1$, and then returns to $x = 0$, on which in spins forever as there are no more results.

This means that computations with infinite failure can be extremely fragile: Oleg's [1] example usage of `>>-` non-terminates if an extra association is added after the `>>-` operator. These situations can generally be avoided by separating out interleaved generation and filtering, which removes the possibility of infinite failure.

Logical conditional

During the process of filtering candidate solutions, we can use standard `MonadPlus` functions such as `guard` to implement deterministic checks against the solution.

However, there is not a good way to conditionalize on finite failure; that is, given a candidate solution, how might we spin off another nondeterministic computation to see if we want to carry on, or try something different? We could use `observeAll` to pull the data out of the Logic monad, and then check if the list is empty, but this defeats composability and isn't very efficient.

The Logic monad defines the `ifte` operator for precisely this case. The expression `ifte expr th el` is equivalent to `expr >>= th` if expr succeeds with at least one result, and equivalent to `el` if it does not. The intuition is similar to Haskell's `if..then..else` construct, except that the results of `expr` are made available to `th` if it succeeds. Oleg [1] suggests that `ifte` is especially useful for "explaining failure" and applying heuristics; an operator with identical semantics exists in Prolog with the name "soft cut."

Pruning

In many computations, we only care about the first result we find: we may be looking for a single counterexample. While laziness generally ensures that results become accessible "as they are needed", thus cutting down on wasteful computation, the ability to tell a computation that only the first result is needed means that we can free any memory that was being used to keep track of backtracking. We can indicate this using the `once` operator.

Further reading

For example uses of `ifte` and `once`, I highly recommend checking out the paper [1] which defines this monad. If you are not interested in how the Logic monad is implemented, skip the sections about `msplit`.

The `stream-monad` package [2, 3] implements some later research by Oleg on nondeterministic computations. It's a bit simpler than the Logic monad, and provides more fair interleaving. It is not to be confused with the `stream-fusion` package, which also provides `Control.Monad.Stream`.

Luke Palmer has implemented the Omega monad [4], which is the List monad but does breadth-first search instead of depth-first search, and Sebastian Palmer has implemented the Level monad [5], which does breadth-first search as well as iterative deepening.

The Prompt Monad

The Prompt monad [6], and the associated typeclass `MonadPrompt`, gives us the ability to restrict side effects with the type system, while giving us the flexibility

to change flow control based on values that are normally stuck in the IO monad. It is provided by the `MonadPrompt` package.

The problem the Prompt monad solves requires a little motivation. Purity is a "big idea" in Haskell: it means that using just a type we can reason about the side effects a computation may have. Haskellers are encouraged to use as restrictive a type as possible to get the job done: pure code is best, and the smaller the monad stack the better.

However, the IO monad remains the elephant in the room: we "need" it to make side effects in the outside world, but the moment we put code in the IO monad we are letting it do anything it wants (such as fire ze missiles.) A simple workaround is to define a data type that encodes effectful actions we want to permit which our pure code gives to a small function in the IO monad responsible for actually executing these effects. The type system then guarantees that only those actions can be executed, and all is well in paradise... that is, until we want our code to respond to the external environment as well. The Prompt monad, as its name suggests, solves precisely this problem.

The Prompt monad is also an example of how monads can introduce an abstraction layer. We can swap out IO with some testing jig that generates and receives information as if it were the environment, without changing any of the code in the Prompt monad (this is quite difficult to do in traditional impure languages, which usually resort to grody metaprogramming tricks.)

The Prompt API

For the Logic monad, we were able to appeal to our experience manipulating lists to figure out how to make the corresponding operations for the Logic monad. Unfortunately, the Prompt monad has no such equivalence; fortunately, the most commonly used portion of the monad is very short, as seen in Listing 11.

```
class Monad m => MonadPrompt p m where
    prompt :: p a -> m a
```

```haskell
data Prompt p r
instance MonadPrompt p (Prompt p)
instance MonadPrompt p (PromptT p m)

runPrompt :: (forall a. p a -> a) -> Prompt p r -> r
runPromptM :: Monad m => (forall a. p a -> m a) -> Prompt p r -> m r
```

Listing 11: Prompt monad API

As with any monad API, there are three sections: the first is the most general monad typeclass which defines any special operations that are intrinsic to the monad; the second are instances that you'd actually use in your program; and the last are functions that let you actually run the monad.

The typeclass `MonadPrompt` defines a single function: `prompt`, which takes as an argument `p a` representing the "request" being made, and returns the "response" inside the Prompt monad. We'll define values to pass as `p a` using generalized abstract data types, in order have more fine-tuned control over what `a`, the return type, is; when utilizing `Prompt` or `PromptT`, we pass simply `p` to indicate what types of requests the Prompt monad services. (For those of you paying attention to kinds, this might seem slightly unusual, since the kind of `p` is `* -> *`, so `Prompt` actually has kind `(* -> *) -> * -> *`, a type usually seen in monad transformers).

The functions `runPrompt` and `runPromptM` take a function that converts our "requests" either into pure values or values inside a monad of the user's choosing, and run the prompt monad with that function. The `forall` in their type signatures indicate a rank-2 type, which is used in order to let the functions `p a -> a` and `p a -> m a` range over multiple values of `a` without getting "stuck" to a particular `a` once we've passed it to `runPrompt`. If this description seems confusing, don't worry; we'll be looking at the form of `p a` and the function `p a -> m a` closely in the following sections.

Generalized abstract data types

Idiomatic use of the Prompt monad involves generalized abstract data types (hitherto referred to as GADTs), so you'll need the `GADTs` GHC language extension. GADTs are an extension to normal abstract data types (the types you define using the `data` keyword) that allow richer return types for the data constructors—applications range from generic pretty-printing to strongly-typed evaluators. [7] In the case of the Prompt monad, we will define a GADT that we'll use to request values from the outside the Prompt monad, and define a function of type

`Request a -> IO a` to serve these requests. Without GADTs, we have no way of restricting `a` into a more specific type[1]; as Ryan Ingram says, "the GADT is serving as a witness of the type of response wanted by the [program]." [8]

To give you a feel for how GADTs are a superset of normal abstract data types, consider the following equivalent pieces of code in Listing 12. In the GADT example, we make explicit the type signature of the data constructor. Note that they still are data constructors, so we can still use pattern matching, we can't return any old value (it has to be of type `GADTExample ?`, where `?` is some type, `a` in this case), and we don't need to make "definitions" for `Zero`, `One` or `Two`, but it

[1] Although, a less beautiful alternate implementation could be made with existential types.


```
data NormalExample a = Zero' | One' a | Two' a a

data GADTExample a where
    Zero :: GADTExample a
```

```
    One :: a -> GADTExample a
    Two :: a -> a -> GADTExample a
```

Listing 12: Syntax comparison for GADTs

does make clear their functional nature; for example, `Two` is curried and the type of `Two True` is `Bool -> GADTExample Bool`.

As mentioned before, GADTs allow richer return types, i.e. the return type need not be `GADTExample a`. Listing 13 contains an actual GADT that will serve as the basis for our `Prompt` example.

```
data Request a where
    Echo :: String -> Request ()
    GetLine :: Request (Maybe String)
    GetTime :: Request UTCTime
```

Listing 13: GADT for the Prompt monad

These data constructors look suspiciously like functions that "echo" and "get a line," returning a value in some sort of `Request` wrapper. And indeed, we've used the GADT in order to indicate both the input types and the output types of an effectful procedure. However, this type doesn't tell us how to go from input to output.

Running the monad

We need to define a function that converts `Request`s, which are plain old data types into actual side-effects and values behind the scenes. The definition in Listing 14 is fairly straightforward, although we use some exception handling capabilities to represent lines retrieved from standard input as either `Just String` or `Nothing`, which indicates an end-of-file. Notice that `a` takes on multiple values depending on what `Request` is pattern-matched; for `Echo s` and `GetLine` it is `String`, but for `GetTime` it is `UTCTime`—this is a feature of GADTs.

With handler function and GADT in hand, we can now write the monadic prompt code in Listing 15 and execute it. `prompt` has the type `p a -> Prompt p a`.


```
handleIO :: Request a -> IO a
handleIO (Echo s) = putStrLn s
```

```haskell
handleIO GetLine = catchJust
    (guard . isEOFError)
    (Just <$> getLine)
    (const (return Nothing))
handleIO GetTime = getCurrentTime
```

Listing 14: Handler function

In our example, `p` is `Request`, and `a` is the return value of `Request`. We pass `prompt` our `Request`s, and we get the result of the request back, `handleIO` pulling the strings behind the scenes.

```haskell
runCat :: IO ()
runCat = runPromptM handleIO cat

cat :: Prompt Request ()
cat = do
    line <- prompt GetLine
    maybe (return ()) (\x -> prompt (Echo x) >> cat) line
```

Listing 15: Implementation of `cat`

Even better, since the monadic code makes no mention of the IO monad, we can easily swap out `handleIO` for some other function, for example, one that replays a transcript, as is seen in Listing 16. Instead of a handler function that shifts from the GADT to the IO monad, will shift to the RWS (Reader, Writer, State) monad to help us thread the transcript through operation and collect the results of this computation.[2]

Further reading

The Prompt monad doesn't have to only be used for a command line interface; Felipe Lessa explores several possiblities for hooking up the Prompt monad to GTK. [9]

[2] Disregard any naysayers claiming this is merely a very convoluted way of implementing `id` for `[String] -> [String]`.

```
type Input  = [String]
type Output = [String]

handleRWS :: Request a -> RWS r Output Input a
handleRWS (Echo s) = tell (return s)
handleRWS GetLine = do
    lines <- get
    if null lines
        then return Nothing
        else do
```

```haskell
            put (tail lines)
            return (Just (head lines))

rwsCat :: RWS r Output Input ()
rwsCat = runPromptM handleRWS cat

simulateCat :: Input -> Output
simulateCat input = snd $ evalRWS rwsCat undefined input
```

Listing 16: Pure simulation of `cat`

While the Prompt monad works well with small programs, on the order of Unix utilities, it's unclear how well this monad scales to larger user applications that may be graphical, may have many more than a dozen ways for the user to interact with the application, or may require asynchronous interaction. This is an area of active research: possible places to look include functional reactive programming (c.f. spreadsheets) and arrows.

The Failure Monad

The Failure monad [10] is not a monad per se, but a class `MonadFailure` for monads that can fail, possibly with error information. It is provided by the `control-monad-failure` and `control-monad-failure-mtl` packages (see page 21 for an explanation). There are also Applicative and Functor versions, although we will not discuss them here. The package grew out of a frustration with the variety of error handling mechanisms that abounded between libraries on Hackage; given any function that may fail, you may get back a value wrapped in any of `Maybe`, `Either`, `ErrorT`, a custom error type, or perhaps get an exception, which cannot be caught until the IO monad. The dream is to automatically compose multiple calls to errorful functions.

The Failure monad doesn't quite fulfill the dream, but it's an important step in the right direction. The fact that it a typeclass means that code can be written for some generic monad that may fail, and then the user of the code can instantiate whichever monad they wish to handle the error. If you are a library writer, you should strongly consider publishing an interface that is merely a generic `MonadFailure`: with some extra restrictions on the type, this interface can be made exactly backwards-compatible. And anyone, application writer or library writer, can use `MonadFailure` delay any decision about which specific error wrapper to use until the error needs to be handled. This is good style and improves composability.

The Failure API

The Failure API is extremely simple, as shown in Listing 17, because it doesn't need to define any functions to run the monad; any monad that has a `MonadFailure` instance will have its own functions for running the monad. The single function `failure` takes an argument of the error type `e`, and can be used as any type within the monad.


```
class Failure e m where
    failure :: e -> m v
class (Monad m, Applicative m, Failure e m) => MonadFailure e m
```

Listing 17: Failure monad API

Conversions

The `MonadFailure` typeclass has two type parameters: `e`, which is the type of the data actually holding information about your error (`String`, `Error`, etc), and `m`, which is the actual monad that can fail. There are lots of ways to express failure in Haskell, [11] so we'll demonstrate how to convert them to use the MonadFailure typeclass.

Consider the simple implementation of `safeHead` in Listing 18. The fact that this

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

Listing 18: `head` with `Maybe`

function emits no error information means we have some latitude when genericizing it. Listing 19 is one possible translation. Notice that `safeHeadFailure` can be

```
safeHeadFailure :: MonadFailure String m => [a] -> m a
safeHeadFailure [] = failure "empty list"
safeHeadFailure (x:xs) = return x

safeHead' :: [a] -> Maybe a
```

```
safeHead' = safeHeadFailure
```

Listing 19: `head` with `MonadFailure`

instantiated into the original, as is shown by `safeHead'`. You can instantiate any value of class `MonadFailure` into the `Maybe` monad without regard to the type of `e`, which is somewhat arbitrarily chosen to be `String` in this case.

Next, we'll consider converting from a monad that does retain error information, `Either`, in Listing 20. This code is written in a different style than the `Maybe` example; namely, it eschews explicit constructors of `Either` in favor of


```
safeHeadPair :: [a] -> [b] -> Either String (a, b)
safeHeadPair [] [] = fail "both lists empty"
safeHeadPair [] _  = fail "first list empty"
safeHeadPair _ []  = fail "second list empty"
safeHeadPair (x:xs) (y:ys) = return (x, y)
```

Listing 20: Pair of heads of list with `Either String`

the monad functions `fail` and `return`. In fact, the code could work under any monadic type, although care should be taken since many monads don't have a meaningful implementation of `fail` and thus default to bottom, and additionally `Either` is not a monad by default; you must import `Control.Monad.Error` to make `Error e => Either e` a monad (and `String` just happens to be an instance of `Error`—if this seems convoluted, it's because it is). The conversion to `MonadFailure` is a straight-forward replacement of `fail` with `failure`, as seen in Listing 21.[3]

```
safeHeadPair' :: MonadFailure String m => [a] -> [b] -> m (a, b)
safeHeadPair' [] [] = failure "both lists empty"
safeHeadPair' [] _  = failure "first list empty"
safeHeadPair' _ []  = failure "second list empty"
safeHeadPair' (x:xs) (y:ys) = return (x, y)
```

Listing 21: Pair of heads of list with `Either String`

The final example in Listing 22 is a bit longer, and serves to illuminate the style of monadic programming that `MonadFailure` encourages, as well as demonstrate that using `Failure` can be useful even if you are not a library writer. We develop a simple system for checking out items from a library: these items are created with a checkout date and a due date. The `renew` function lets someone push their due date later, but only if the book hadn't already expired (in which case it errors).

There are a few features of note: we use `type` to explicitly create a monad stack called `TimeMonad`, which has the Reader monad capability to determine the current time, as well as the Error monad transformer, which permits us to error out. `MyError` is a userland data type that encodes errors that this application may emit; in practice the type would be much longer (a small 500-line application I wrote

[3] Since `MonadFailure` requires the `Monad` instance on `m`, `fail` would probably work in the case of `e` being `String`.

```haskell
type TimeMonad  = ErrorT MyError (Reader UTCTime)

data MyError = ExpiredError
             | MyParseError ParseError -- see Marshalling
             | MiscError String
    deriving (Show)
instance Error MyError where
    noMsg = MiscError "Unknown error"
    strMsg s = MiscError s

data Checkout = Checkout
    { checkoutName :: String
    , checkoutTime :: UTCTime
    , checkoutDue  :: UTCTime
    }

checkoutLength :: Checkout -> NominalDiffTime
checkoutLength c = diffUTCTime (checkoutDue c) (checkoutTime c)
```

```haskell
shiftCheckoutTime :: Checkout -> UTCTime -> Checkout
shiftCheckoutTime c newTime = c
    { checkoutTime = newTime
    , checkoutDue  = addUTCTime (checkoutLength c) newTime
    }

renew :: Checkout -> TimeMonad Checkout
renew c = do
    curTime <- ask
    when (checkoutDue c < curTime) $ throwError ExpiredError
    return (shiftCheckoutTime c curTime)
```

Listing 22: A simple checkout renewal system contained eighteen error constructors), and permits writing code in an "exception throwing" style without actually using asynchronous or imprecise exceptions: a code that throws an error bubbles up until some level of execution deals with it.

The Error monad is quite a heavy hammer, and I have initially written code in the Maybe monad, only to have to go on a search, replace and typecheck hunt when I realize `Nothing` isn't actually sufficient information when there are several layers of code in the Maybe monad, all of which could have resulted in this error. With the Failure monad I can build in this capability from the start, but use it with the simpler Maybe monad interface unless I need detailed information about the error.

```
renew' :: (MonadReader UTCTime m, MonadFailure MyError m) =>
          Checkout -> m Checkout
renew' c = do
    curTime <- ask
    when (checkoutDue c < curTime) $ failure ExpiredError
    return (shiftCheckoutTime c curTime)
```

Listing 23: Implementation using typeclasses

Listing 23 contains two changes: the first is familiar; we've changed `throwError` to `failure`. The other is the changed function signature. The original implementation was tied to the `TimeMonad`; the new code is more generic because all the type requires is that the monad `m` have the `MonadReader UTCTime` and the `MonadFailure MyError` "capabilities"; the actual `m` we pass in could be arbitrarily more powerful but the type signature enforces that the resulting code will only use those "capabilities." Additionally, the new type signature expresses the fact that the Reader monad and the Failure monad commute.

Marshalling

The `failure` package is still fairly nascent, and as such you are unlikely to see third-party libraries exporting functions with it... yet. In the meantime, `Control.Failure` exports the `try` function (part of the `Try` typeclass) which permits us to easily marshal values in other Monads into another `MonadFailure` form.[4] It's interface is described in Listing 24.

If the input type `m` and output type `m'` are the same, try acts as an identity, as shown in Listing 25[5].

[4] It works for applicatives too, as seen in the type signature.

[5] As of writing, the example for Either requires an import of `Control.Applicative` and, for at

```
class Try m where
    type Error m
    try :: ApplicativeFailure (Error m) m' => m a -> m' a
```

Listing 24: Try API

```
maybeVal :: Maybe Int
maybeVal = try $ Just 3

eitherVal :: Either String ()
eitherVal = try $ Left "error"
```

Listing 25: Try as identity

However, in many cases, what we'd really like to do is take an arbitrary error type from some third-party library and convert it into our own, application specific error type. One simple way to do this is to have a wrapper constructor inside your error data type, and defer handling the error to your global error handling code.

Curiously enough, `MyError` from the `TimeMonad` example has a constructor defined just for Parsec! In Listing 26, we take a `ParseError` from Parsec and place it into `MyParseError`, which we defined previously in Listing 22. The instance "lifts"

```haskell
instance Failure MyError m => Failure ParseError m where
    failure e = failure (MyParseError e)

theirParse :: Parser a -> String -> Either ParseError a
theirParse parser s = parse parser "" s

myParse :: (MonadFailure MyError m, Failure ParseError m) =>
           Parser a -> String -> m a
myParse parser s = try $ theirParse parser s
```

Listing 26: Implementation using typeclasses

the failure from `Either ParseError` into the more general `Failure MyError m => m`. There is one oddity in this code, which is the specification of `Failure ParseError m` in the signature: without it, `m` is overly general and results in overlapping instances.

least mtl, an orphan instance of Applicative for Either.

If you instantiate `myParse` anywhere else in the module, Haskell will be able to infer the correct type, but otherwise you need that extra restriction.

We should note that there is a namespace collision between failure and parsec on `try`, so we suggest keeping your parsec code in a one module and your failure code in another.

There is another way to pass around errors from arbitrary third parties: instead of defining an error type, define an error typeclass and write instances of it for every third-party error type you want to support. You don't even need `try`; any errorable type will cleanly "cast" into the more generalized typeclass. The downside of this approach is that this typeclass will have to support any type of operation you may want to do: it is the only interface you get for accessing error information.

Addendum

The Failure monad publishes two versions of its module: `Control.Monad.Failure` and `Control.Monad.Failure.MTL`. This stems from the fact that there are three widely recognized monad libraries inside Haskell: `mtl`, which comes by default with GHC; `transformers`, which defines monads in terms of transformers on top of the Identity monad; and `monadLib`, Galois' brainchild and similar to transformers. Each of these defines important monadic types and instances.

If you try mixing two libraries together, even indirectly (from an external library that imports a different monad library, you'll notice two things: first, you'll have numerous ambiguous occurrences of constructors from the monads, since both library will attempt to export it's own, and second, you'll have overlapping instances as each library attempts to define its own. Furthermore, functions from the one library will refuse to take data from the other: defined in separate modules, they are different types.

Practically speaking, you should pick one of these libraries and stick with it. This article is written with `mtl`, and should be translatable to another Monad library with a little coaxing.

Acknowledgments

I'd like to thank the SIPB members who originally introduced me to this wonderful language, Brent Yorgey, who with his wonderful Typeclassopedia convinced me to write something for The Monad Reader and who was immensely helpful in the initial review process, as well as the denizens of `#haskell` who've put up with my ruminations and questions.

In particular, I'd like to thank Dan (dolio) Doel, Walt (BMeph) Rorie-Baety, Berengal, David (dmhouse) House, Albert (monochrom) Lai, lpsmith, elcerdo-

querie, Ricardo Herrmann, T_S_, jolanda_ypsilon, Sebastian Fischer, Daniel (copumpkin) Peebles, kmc, Andy (aavogt) Vogt, Wei Hu.

References

[1] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In **ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming**, pages 192–203. ACM, New York, NY, USA (2005). `http://okmij.org/ftp/papers/LogicT.pdf`.

[2] Oleg Kiselyov. Simple fair and terminating backtracking monad transformer. `http://okmij.org/ftp/Computation/monads.html#fair-bt-stream`.

[3] Oleg Kiselyov and Sebastian Fischer. stream-monad: Simple, fair and terminating backtracking monad. `http://hackage.haskell.org/package/stream-monad`.

[4] Luke Palmer. control-monad-omega: A breadth-first list monad. `http://hackage.haskell.org/package/control-monad-omega`.

[5] Sebastian Fischer. level-monad: Non-Determinism Monad for Level-Wise Search.

http://hackage.haskell.org/package/level-monad.

[6] Ryan Ingram and Bertram Felgenhauer. MonadPrompt: MonadPrompt, implementation and examples. http://hackage.haskell.org/package/MonadPrompt.

[7] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In **OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, pages 21–40. ACM, New York, NY, USA (2005). http://research.microsoft.com/en-us/um/people/akenn/generics/gadtoop.pdf.

[8] Ryan Ingram. An interesting monad: "Prompt". http://www.mail-archive.com/haskell-cafe@haskell.org/msg33040.html.

[9] Felipe Lessa. MonadPrompt + Gtk2Hs = ? http://www.mail-archive.com/haskell-cafe@haskell.org/msg36256.html.

[10] Pepe Iborra, Michael Snoyman, and Nicolas Pouillard. control-monad-failure: A class for monads which can fail with an error. http://hackage.haskell.org/package/control-monad-failure.

[11] Eric Kidd. 8 ways to report errors in Haskell. http://www.randomhacks.net/articles/2007/03/10/haskell-8-ways-to-report-errors.

Appendix

```
instance Enum TuringState where
    succ Halt = State minBound
    succ (State x) = State (succ x)
    pred (State x) | x == minBound = Halt
                   | otherwise     = State (pred x)
    toEnum 0 = Halt
    toEnum n = State $ toEnum (n-1)
    fromEnum Halt = 0
    fromEnum (State x) = 1 + fromEnum x
instance Bounded TuringState where
    minBound = Halt
    maxBound = State maxBound
instance Ix TuringState where
```

```haskell
range (n, m) = [n..m]
index (Halt, m) Halt = 0
index (Halt, State m) (State x) = 1 + index (minBound, m) x
index (State n, State m) (State x) = index (n, m) x
inRange (Halt, _) Halt = True
inRange (Halt, State m) (State x) = inRange (minBound, m) x
inRange (State n, State m) (State x) = inRange (n, m) x
inRange _ _ = False
```

Listing 27: Enum, Bounded and Ix instances for TuringState

The Monad.Reader

23

```
instance Show Tape where
    show (Tape left cur right) = intercalate " " symbols
        where symbols = "..." : symbols' ++ ["..."]
              symbols' =
                    reverse (lshow 3 left) ++
                    ['*' : show cur ++ "*"] ++
                    lshow 17 right
              lshow n xs = map show $ take n xs
instance Show TuringTransition where
    show logic | null $ observeAll logic = "*undefined*"
               | otherwise = intercalate ", " strings
                    where strings = [show a, show b, show c]
                          (a, b, c) = observe logic
```

Listing 28: Show instances for `Tape` and `TuringTransition`