Chapter 15. Programming with monads

Table of Contents

Golfing practice: association lists

Web clients and servers often pass information around as a simple textual list of key-value pairs. No comments

```
name=Attila+%42The+Hun%42&occupation=Khan
```

No comments

The encoding is named `application/x-www-form-urlencoded`, and it's easy to understand. Each key-value pair is separated by an “`&`” character. Within a pair, a key is a series of characters, followed by an “`=`”, followed by a value. 2 comments

We can obviously represent a key as a `String`, but the HTTP specification is not clear about whether a key must be followed by a value. We can capture this ambiguity by representing a value as a `Maybe String`. If we use `Nothing` for a value, then there was no value present. If we wrap a string in `Just`, then there was a value. Using `Maybe` lets us distinguish between “no value” and “empty value”. 5 comments

Haskell programmers use the name *association list* for the type `[(a, b)]`, where we can think of each element as an association between a key and a value. The name originates in the Lisp community, where it's usually abbreviated as an *alist*. We could thus represent the above string as the following Haskell value. 2 comments

```
-- file: ch15/MovieReview.hs
    [("name",       Just "Attila \"The Hun\""),
     ("occupation", Just "Khan")]
```

4 comments

In the section called “Parsing an URL-encoded query string”, we'll parse an `application/x-www-form-urlencoded` string, and represent the result as an alist of `[(String, Maybe String)]`. Let's say we want to use one of these alists to fill out a data structure. No comments

```
-- file: ch15/MovieReview.hs
data MovieReview = MovieReview {
      revTitle :: String
    , revUser :: String
    , revReview :: String
    }
```

2 comments

We'll begin by belabouring the obvious with a naive function. 3 comments

```
-- file: ch15/MovieReview.hs
simpleReview :: [(String, Maybe String)] -> Maybe MovieReview
simpleReview alist =
  case lookup "title" alist of
    Just (Just title@(_:_)) ->
      case lookup "user" alist of
        Just (Just user@(_:_)) ->
          case lookup "review" alist of
            Just (Just review@(_:_)) ->
                Just (MovieReview title user review)
            _ -> Nothing -- no review
        _ -> Nothing -- no user
    _ -> Nothing -- no title
```

2 comments

It only returns a `MovieReview` if the alist contains all of the necessary values, and they're all non-empty strings. However, the fact that it validates its inputs is its only merit: it suffers badly from the "staircasing" that we've learned to be wary of, and it knows the intimate details of the representation of an alist. No comments

Since we're now well acquainted with the `Maybe` monad, we can tidy up the staircasing. No comments

```
-- file: ch15/MovieReview.hs
maybeReview alist = do
    title <- lookup1 "title" alist
    user <- lookup1 "user" alist
    review <- lookup1 "review" alist
    return (MovieReview title user review)

lookup1 key alist = case lookup key alist of
                      Just (Just s@(_:_)) -> Just s
                      _ -> Nothing
```

7 comments

Although this is much tidier, we're still repeating ourselves. We can take advantage of the fact that the `MovieReview` constructor acts as a normal, pure function by *lifting* it into the monad, as we discussed in the section called "Mixing pure and monadic code". No comments

```
-- file: ch15/MovieReview.hs
liftedReview alist =
    liftM3 MovieReview (lookup1 "title" alist)
                       (lookup1 "user" alist)
                       (lookup1 "review" alist)
```

4 comments

We still have some repetition here, but it is dramatically reduced, and also more difficult to remove. 9 comments

Generalised lifting

Although using `liftM3` tidies up our code, we can't use a `liftM`-family function to solve this sort of problem in general, because they're only defined up to `liftM5` by the standard libraries. We could write variants up to whatever number we pleased, but that would amount to drudgery. 3 comments

If we had a constructor or pure function that took, say, ten parameters, and decided to stick with the standard libraries you might think we'd be out of luck. 2 comments

Of course, our toolbox isn't yet empty. In `Control.Monad`, there's a function named `ap` with an interesting type signature. 5 comments

```
ghci> :m +Control.Monad
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

No comments

You might wonder who would put a single-argument pure function inside a monad, and why. Recall, however, that *all* Haskell functions really take only one argument, and you'll begin to see how this might relate to the `MovieReview` constructor. No comments

```
ghci> :type MovieReview
MovieReview :: String -> String -> String -> MovieReview
```

No comments

We can just as easily write that type as `String -> (String -> (String -> MovieReview))`. If we use plain old `liftM` to lift `MovieReview` into the `Maybe` monad, we'll have a value of type `Maybe (String -> (String -> (String -> MovieReview)))`. We can now see that this type is suitable as an argument for `ap`, in which case the result type will be `Maybe (String -> (String -> MovieReview))`. We can pass this, in turn, to `ap`, and continue to chain until we end up with this definition. 8 comments

```
-- file: ch15/MovieReview.hs
apReview alist =
    MovieReview `liftM` lookup1 "title" alist
                   `ap` lookup1 "user" alist
                   `ap` lookup1 "review" alist
```

5 comments

We can chain applications of `ap` like this as many times as we need to, thereby bypassing the `liftM` family of functions. 2 comments

Another helpful way to look at `ap` is that it's the monadic equivalent of the familiar `($)` operator: think of pronouncing `ap` as *apply*. We can see this clearly when we compare the type signatures of the two functions. 4 comments

```
ghci> :type ($)
($) :: (a -> b) -> a -> b
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

2 comments

In fact, `ap` is usually defined as either `liftM2 id` or `liftM2 ($)`. 10 comments

Looking for alternatives

Here's a simple representation of a person's phone numbers. No comments

```
-- file: ch15/VCard.hs
data Context = Home | Mobile | Business
               deriving (Eq, Show)

type Phone = String

albulena = [(Home, "+355-652-55512")]

nils = [(Mobile, "+47-922-55-512"), (Business, "+47-922-12-121"),
        (Home, "+47-925-55-121"), (Business, "+47-922-25-551")]
        twalumba = [(Business, "+260-02-55-5121")]
```

No comments

Suppose we want to get in touch with someone to make a personal call. We don't want their business number, and we'd prefer to use their home number (if they have one) instead of their mobile number. 3 comments

```
        -- file: ch15/VCard.hs
        onePersonalPhone :: [(Context, Phone)] -> Maybe Phone
        onePersonalPhone ps = case lookup Home ps of
                                Nothing -> lookup Mobile ps
                                Just n -> Just n
```

2 comments

Of course, if we use `Maybe` as the result type, we can't accommodate the possibility that someone might have more than one number that meet our criteria. For that, we switch to a list. No comments

```
        -- file: ch15/VCard.hs
        allBusinessPhones :: [(Context, Phone)] -> [Phone]
        allBusinessPhones ps = map snd numbers
            where numbers = case filter (contextIs Business) ps of
                              [] -> filter (contextIs Mobile) ps
                              ns -> ns

        contextIs a (b, _) = a == b
```

1 comment

Notice that these two functions structure their `case` expressions similarly: one alternative handles the case where the first lookup returns an empty result, while the other handles the non-empty case. No comments

```
        ghci> onePersonalPhone twalumba
        Nothing
        ghci> onePersonalPhone albulena
        Just "+355-652-55512"
        ghci> allBusinessPhones nils
        ["+47-922-12-121","+47-922-25-551"]
```

1 comment

Haskell's `Control.Monad` module defines a typeclass, `MonadPlus`, that lets us abstract the common pattern out of our `case` expressions. 1 comment

```
        -- file: ch15/VCard.hs
        class Monad m => MonadPlus m where
           mzero :: m a
           mplus :: m a -> m a -> m a
```

No comments

The value `mzero` represents an empty result, while `mplus` combines two results into one. Here are the standard definitions of `mzero` and `mplus` for `Maybe` and lists. No comments

```
        -- file: ch15/VCard.hs
        instance MonadPlus [] where
           mzero = []
           mplus = (++)

        instance MonadPlus Maybe where
           mzero = Nothing
```

```
    Nothing `mplus` ys  = ys
    xs      `mplus` _ = xs
```

3 comments

We can now use `mplus` to get rid of our `case` expressions entirely. For variety, let's fetch one business and all personal phone numbers. 2 comments

```
-- file: ch15/VCard.hs
oneBusinessPhone :: [(Context, Phone)] -> Maybe Phone
oneBusinessPhone ps = lookup Business ps `mplus` lookup Mobile ps

allPersonalPhones :: [(Context, Phone)] -> [Phone]
allPersonalPhones ps = map snd $ filter (contextIs Home) ps `mplus`
                                 filter (contextIs Mobile) ps
```

8 comments

In these functions, because we know that `lookup` returns a value of type `Maybe`, and `filter` returns a list, it's obvious which version of `mplus` is going to be used in each case. 3 comments

What's more interesting is that we can use `mzero` and `mplus` to write functions that will be useful for *any* `MonadPlus` instance. As an example, here's the standard `lookup` function, which returns a value of type `Maybe`. 1 comment

```
-- file: ch15/VCard.hs
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
lookup _ []                   = Nothing
lookup k ((x,y):xys) | x == k     = Just y
                     | otherwise = lookup k xys
```

No comments

We can easily generalise the result type to any instance of `MonadPlus` as follows. No comments

```
-- file: ch15/VCard.hs
lookupM :: (MonadPlus m, Eq a) => a -> [(a, b)] -> m b
lookupM _ []    = mzero
lookupM k ((x,y):xys)
    | x == k    = return y `mplus` lookupM k xys
    | otherwise = lookupM k xys
```

2 comments

This lets us get either no result or one, if our result type is `Maybe`; all results, if our result type is a list; or something more appropriate for some other exotic instance of `MonadPlus`. 6 comments

For small functions, such as those we present above, there's little benefit to using `mplus`. The advantage lies in more complex code and in code that is independent of the monad in which it executes. Even if you don't find yourself needing `MonadPlus` for your own code, you are likely to encounter it in other people's projects. 2 comments

The name mplus does not imply addition

Even though the `mplus` function contains the text “plus”, you should not think of it as necessarily implying that we're trying to add two values. No comments

Depending on the monad we're working in, `mplus` *may* implement an operation that looks like addition. For example, `mplus` in the list monad is implemented as the `(++)` operator. 1 comment

```
ghci> [1,2,3] `mplus` [4,5,6]
```

```
[1,2,3,4,5,6]
```

No comments

However, if we switch to another monad, the obvious similarity to addition falls away. 1 comment

```
ghci> Just 1 `mplus` Just 2
Just 1
```

2 comments

Rules for working with MonadPlus

Instances of the `MonadPlus` typeclass must follow a few simple rules, in addition to the usual monad rules. No comments

An instance must *short circuit* if `mzero` appears on the left of a bind expression. In other words, an expression `mzero >>= f` must evaluate to the same result as `mzero` alone. 5 comments

```
-- file: ch15/MonadPlus.hs
    mzero >>= f == mzero
```

No comments

An instance must short circuit if `mzero` appears on the *right* of a sequence expression. 3 comments

```
-- file: ch15/MonadPlus.hs
    v >> mzero == mzero
```

3 comments

Failing safely with MonadPlus

When we introduced the `fail` function in the section called “The Monad typeclass”, we took pains to warn against its use: in many monads, it's implemented as a call to `error`, which has unpleasant consequences. No comments

The `MonadPlus` typeclass gives us a gentler way to fail a computation, without `fail` or `error` blowing up in our faces. The rules that we introduced above allow us to introduce an `mzero` into our code wherever we need to, and computation will short circuit at that point. 1 comment

In the `Control.Monad` module, the standard function `guard` packages up this idea in a convenient form. No comments

```
-- file: ch15/MonadPlus.hs
guard        :: (MonadPlus m) => Bool -> m ()
guard True   =  return ()
guard False  =  mzero
```

2 comments

As a simple example, here's a function that takes a number `x` and computes its value modulo some other number `n`. If the result is zero, it returns `x`, otherwise the current monad's `mzero`. 1 comment

```
-- file: ch15/MonadPlus.hs
x `zeroMod` n = guard ((x `mod` n) == 0) >> return x
```

8 comments

Adventures in hiding the plumbing

In the section called "Using the state monad: generating random values", we showed how to use the `State` monad to give ourselves access to random numbers in a way that is easy to use. No comments

A drawback of the code we developed is that it's *leaky*: someone who uses it knows that they're executing inside the `State` monad. This means that they can inspect and modify the state of the random number generator just as easily as we, the authors, can. No comments

Human nature dictates that if we leave our internal workings exposed, someone will surely come along and monkey with them. For a sufficiently small program, this may be fine, but in a larger software project, when one consumer of a library modifies its internals in a way that other consumers are not prepared for, the resulting bugs can be among the hardest of all to track down. These bugs occur at a level where we're unlikely to question our basic assumptions about a library until long after we've exhausted all other avenues of inquiry. No comments

Even worse, once we leave our implementation exposed for a while, and some well-intentioned person inevitably bypasses our APIs and uses the implementation directly, we create a nasty quandary for ourselves if we need to fix a bug or make an enhancement. Either we can modify our internals, and break code that depends on them; or we're stuck with our existing internals, and must try to find some other way to make the change we need. 1 comment

How can we revise our random number monad so that the fact that we're using the `State` monad is hidden? We need to somehow prevent our users from being able to call `get` or `put`. This is not difficult to do, and it introduces some tricks that we'll reuse often in day-to-day Haskell programming. 1 comment

To widen our scope, we'll move beyond random numbers, and implement a monad that supplies unique values of *any* kind. The name we'll give to our monad is `Supply`. We'll provide the execution function, `runSupply`, with a list of values; it will be up to us to ensure that each one is unique. 4 comments

```
-- file: ch15/Supply.hs
runSupply :: Supply s a -> [s] -> (a, [s])
```

No comments

The monad won't care what the values are: they might be random numbers, or names for temporary files, or identifiers for HTTP cookies. No comments

Within the monad, every time a consumer asks for a value, the `next` action will take the next one from the list and give it to the consumer. Each value is wrapped in a `Maybe` constructor in case the list isn't long enough to satisfy the demand. No comments

```
-- file: ch15/Supply.hs
next :: Supply s (Maybe s)
```

2 comments

To hide our plumbing, in our module declaration we only export the type constructor, the execution function, and the `next` action. No comments

```
-- file: ch15/Supply.hs
module Supply
```

```
            (
              Supply
            , next
            , runSupply
            ) where
```

No comments

Since a module that imports the library can't see the internals of the monad, it can't manipulate them. No comments

Our plumbing is exceedingly simple: we use a `newtype` declaration to wrap the existing `State` monad. No comments

```
-- file: ch15/Supply.hs
import Control.Monad.State

newtype Supply s a = S (State [s] a)
```

No comments

The `s` parameter is the type of the unique values we are going to supply, and `a` is the usual type parameter that we must provide in order to make our type a monad. 2 comments

Our use of `newtype` for the `Supply` type and our module header join forces to prevent our clients from using the `State` monad's `get` and `set` actions. Because our module does not export the `S` data constructor, clients have no programmatic way to see that we're wrapping the `State` monad, or to access it. 1 comment

At this point, we've got a type, `Supply`, that we need to make an instance of the `Monad` type class. We could follow the usual pattern of defining `(>>=)` and `return`, but this would be pure boilerplate code. All we'd be doing is wrapping and unwrapping the `State` monad's versions of `(>>=)` and `return` using our `S` value constructor. Here is how such code would look. 2 comments

```
-- file: ch15/AltSupply.hs
unwrapS :: Supply s a -> State [s] a
unwrapS (S s) = s

instance Monad (Supply s) where
    s >>= m = S (unwrapS s >>= unwrapS . m)
    return = S . return
```

5 comments

Haskell programmers are not fond of boilerplate, and sure enough, GHC has a lovely language extension that eliminates the work. To use it, we add the following directive to the top of our source file, before the module header. No comments

```
-- file: ch15/Supply.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

No comments

Usually, we can only automatically derive instances of a handful of standard typeclasses, such as `Show` and `Eq`. As its name suggests, the `GeneralizedNewtypeDeriving` extension broadens our ability to derive typeclass instances, and it is specific to `newtype` declarations. If the type we're wrapping is an instance of any typeclass, the extensions can automatically make our new type an instance of that typeclass as follows. 3 comments

```
-- file: ch15/Supply.hs
    deriving (Monad)
```

2 comments

This takes the underlying type's implementations of `(>>=)` and `return`, adds the necessary wrapping and unwrapping with our `S` data constructor, and uses the new versions of those functions to derive a `Monad` instance for us. 2 comments

What we gain here is very useful beyond just this example. We can use `newtype` to wrap any underlying type; we selectively expose only those typeclass instances that we want; and we expend almost no effort to create these narrower, more specialised types. No comments

Now that we've seen the `GeneralizedNewtypeDeriving` technique, all that remains is to provide definitions of `next` and `runSupply`. No comments

```
-- file: ch15/Supply.hs
next = S $ do st <- get
              case st of
                [] -> return Nothing
                (x:xs) -> do put xs
                             return (Just x)

runSupply (S m) xs = runState m xs
```

5 comments

If we load our module into **ghci**, we can try it out in a few simple ways. No comments

```
ghci> :load Supply
[1 of 1] Compiling Supply           ( Supply.hs, interpreted )
Ok, modules loaded: Supply.
ghci> runSupply next [1,2,3]
Loading package mtl-1.1.0.0 ... linking ... done.
(Just 1,[2,3])
ghci> runSupply (liftM2 (,) next next) [1,2,3]
((Just 1,Just 2),[3])
ghci> runSupply (liftM2 (,) next next) [1]
((Just 1,Nothing),[])
```

5 comments

We can also verify that the `State` monad has not somehow leaked out. No comments

```
ghci> :browse Supply
data Supply s a
next :: Supply s (Maybe s)
runSupply :: Supply s a -> [s] -> (a, [s])
ghci> :info Supply
data Supply s a         -- Defined at Supply.hs:17:8-13
instance Monad (Supply s) -- Defined at Supply.hs:17:8-13
```

2 comments

Supplying random numbers

If we want to use our `Supply` monad as a source of random numbers, we have a small difficulty to face. Ideally, we'd like to be able to provide it with an infinite stream of random numbers. We can get a `StdGen` in the `IO` monad, but we must “put back” a different `StdGen` when we're done. If we don't, the next piece of code to get a `StdGen` will get the same state as we did. This means it will generate the same random numbers as we did, which is potentially catastrophic. No comments

From the parts of the `System.Random` module we've seen so far, it's difficult to reconcile these demands. We can use `getStdRandom`, whose type ensures that when we get a `StdGen`, we put one back. No comments

```
ghci> :type getStdRandom
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

No comments

We can use `random` to get back a new `StdGen` when they give us a random number. And we can use `randoms` to get an infinite list of random numbers. But how do we get both an infinite list of random numbers *and* a new `StdGen`? 3 comments

The answer lies with the `RandomGen` type class's `split` function, which takes one random number generator, and turns it into two generators. Splitting a random generator like this is a most unusual thing to be able to do: it's obviously tremendously useful in a pure functional setting, but essentially never either necessary or provided by an impure language. 2 comments

Using the `split` function, we can use one `StdGen` to generate an infinite list of random numbers to feed to `runSupply`, while we give the other back to the `IO` monad. No comments

```
-- file: ch15/RandomSupply.hs
import Supply
import System.Random hiding (next)

randomsIO :: Random a => IO [a]
randomsIO =
    getStdRandom $ \g ->
        let (a, b) = split g
        in (randoms a, b)
```

2 comments

If we've written this function properly, our example ought to print a different random number on each invocation. 1 comment

```
ghci> :load RandomSupply
[1 of 2] Compiling Supply           ( Supply.hs, interpreted )
[2 of 2] Compiling RandomSupply     ( RandomSupply.hs, interpreted )
Ok, modules loaded: RandomSupply, Supply.
ghci> (fst . runSupply next) `fmap` randomsIO

<interactive>:1:17:
    Ambiguous occurrence `next'
    It could refer to either `Supply.next', imported from Supply at RandomSupply.hs:4:0-12
                                          (defined at Supply.hs:32:0)
                          or `System.Random.next', imported from System.Random
ghci> (fst . runSupply next) `fmap` randomsIO

<interactive>:1:17:
    Ambiguous occurrence `next'
    It could refer to either `Supply.next', imported from Supply at RandomSupply.hs:4:0-12
                                          (defined at Supply.hs:32:0)
                          or `System.Random.next', imported from System.Random
```

11 comments

Recall that our `runSupply` function returns both the result of executing the monadic action and the unconsumed remainder of the list. Since we passed it an infinite list of random numbers, we compose with `fst` to ensure that we don't get drowned in random numbers when **ghci** tries to print the result. No comments

Another round of golf

The pattern of applying a function to one element of a pair, and constructing a new pair with the other original element untouched, is common enough in Haskell code that it has been turned into standard code. No comments

In the `Control.Arrow` module are two functions, `first` and `second`, that perform this operation. No comments

```
ghci> :m +Control.Arrow
ghci> first (+3) (1,2)
(4,2)
ghci> second odd ('a',1)
('a',True)
```

No comments

(Indeed, we already encountered `second`, in the section called “JSON typeclasses without overlapping instances”.) We can use `first` to golf our definition of `randomsIO`, turning it into a one-liner. 1 comment

```
-- file: ch15/RandomGolf.hs
import Control.Arrow (first)

randomsIO_golfed :: Random a => IO [a]
randomsIO_golfed = getStdRandom (first randoms . split)
```

4 comments

Separating interface from implementation

In the previous section, we saw how to hide the fact that we're using a `State` monad to hold the state for our `Supply` monad. No comments

Another important way to make code more modular involves separating its *interface*—what the code can do—from its *implementation*—how it does it. No comments

The standard random number generator in `System.Random` is known to be quite slow. If we use our `randomsIO` function to provide it with random numbers, then our `next` action will not perform well. No comments

One simple and effective way that we could deal with this is to provide `Supply` with a better source of random numbers. Let's set this idea aside, though, and consider an alternative approach, one that is useful in many settings. We will separate the actions we can perform with the monad from how it works using a typeclass. 2 comments

```
-- file: ch15/SupplyClass.hs
class (Monad m) => MonadSupply s m | m -> s where
    next :: m (Maybe s)
```

No comments

This typeclass defines the interface that any supply monad must implement. It bears careful inspection, since it uses several unfamiliar Haskell language extensions. We will cover each one in the sections that follow. No comments

Multi-parameter typeclasses

How should we read the snippet `MonadSupply s m` in the typeclass? If we add parentheses, an equivalent expression is `(MonadSupply s) m`, which is a little clearer. In other words, given some type variable `m` that is a `Monad`, we can make it an instance of the typeclass `MonadSupply s`. unlike a regular typeclass, this one has a *parameter*. 11 comments

As this language extension allows a typeclass to have more than one parameter, its name is

MultiParamTypeClasses. The parameter s serves the same purpose as the Supply type's parameter of the same name: it represents the type of the values handed out by the next function. 1 comment

Notice that we don't need to mention (>>=) or return in the definition of MonadSupply s, since the type class's context (superclass) requires that a MonadSupply s must already be a Monad. 4 comments

Functional dependencies

To revisit a snippet that we ignored earlier, | m -> s is a *functional dependency*, often called a *fundep*. We can read the vertical bar | as “such that”, and the arrow -> as “uniquely determines”. Our functional dependency establishes a *relationship* between m and s. 2 comments

The availability of functional dependencies is governed by the FunctionalDependencies language pragma. 1 comment

The purpose behind us declaring a relationship is to help the type checker. Recall that a Haskell type checker is essentially a theorem prover, and that it is conservative in how it operates: it insists that its proofs must terminate. A non-terminating proof results in the compiler either giving up or getting stuck in an infinite loop. 3 comments

With our functional dependency, we are telling the type checker that every time it sees some monad m being used in the context of a MonadSupply s, the type s is the only acceptable type to use with it. If we were to omit the functional dependency, the type checker would simply give up with an error message. 3 comments

It's hard to picture what the relationship between m and s really means, so let's look at an instance of this typeclass. No comments

```
-- file: ch15/SupplyClass.hs
import qualified Supply as S

instance MonadSupply s (S.Supply s) where
    next = S.next
```

4 comments

Here, the type variable m is replaced by the type S.Supply s. Thanks to our functional dependency, the type checker now knows that when it sees a type S.Supply s, the type can be used as an instance of the typeclass MonadSupply s. No comments

If we didn't have a functional dependency, the type checker would not be able to figure out the relationship between the type parameter of the class MonadSupply s and that of the type Supply s, and it would abort compilation with an error. The definition itself would compile; the type error would not arise until the first time we tried to use it. 1 comment

To strip away one final layer of abstraction, consider the type S.Supply Int. Without a functional dependency, we could declare this an instance of MonadSupply s. However, if we tried to write code using this instance, the compiler would not be able to figure out that the type's Int parameter needs to be the same as the typeclass's s parameter, and it would report an error. No comments

Functional dependencies can be tricky to understand, and once we move beyond simple uses, they often prove difficult to work with in practice. Fortunately, the most frequent use of functional dependencies is in situations as simple as ours, where they cause little trouble. 7 comments

Rounding out our module

If we save our typeclass and instance in a source file named SupplyClass.hs, we'll need to add a module header such as the following. No comments

```
-- file: ch15/SupplyClass.hs
{-# LANGUAGE FlexibleInstances, FunctionalDependencies,
             MultiParamTypeClasses #-}

module SupplyClass
    (
      MonadSupply(..)
    , S.Supply
    , S.runSupply
    ) where
```

1 comment

The `FlexibleInstances` extension is necessary so that the compiler will accept our instance declaration. This extension relaxes the normal rules for writing instances in some circumstances, in a way that still lets the compiler's type checker guarantee that it will terminate. Our need for `FlexibleInstances` here is caused by our use of functional dependencies, but the details are unfortunately beyond the scope of this book. No comments

How to know when a language extension is needed

If GHC cannot compile a piece of code because it would require some language extension to be enabled, it will tell us which extension we should use. For example, if it decides that our code needs flexible instance support, it will suggest that we try compiling with the `-XFlexibleInstances` option. A `-X` option has the same effect as a `LANGUAGE` directive: it enables a particular extension. 3 comments

Finally, notice that we're re-exporting the `runSupply` and `Supply` names from this module. It's perfectly legal to export a name from one module even though it's defined in another. In our case, it means that client code only needs to import the `SupplyClass` module, without also importing the `Supply` module. This reduces the number of “moving parts” that a user of our code needs to keep in mind. No comments

Programming to a monad's interface

Here is a simple function that fetches two values from our `Supply` monad, formats them as a string, and returns them. 5 comments

```
-- file: ch15/Supply.hs
showTwo :: (Show s) => Supply s String
showTwo = do
  a <- next
  b <- next
  return (show "a: " ++ show a ++ ", b: " ++ show b)
```

2 comments

This code is tied by its result type to our `Supply` monad. We can easily generalize to any monad that implements our `MonadSupply` interface by modifying our function's type. Notice that the body of the function remains unchanged. No comments

```
-- file: ch15/SupplyClass.hs
showTwo_class :: (Show s, Monad m, MonadSupply s m) => m String
showTwo_class = do
  a <- next
  b <- next
  return (show "a: " ++ show a ++ ", b: " ++ show b)
```

1 comment

The reader monad

The `State` monad lets us plumb a piece of mutable state through our code. Sometimes, we would like to be able to pass some *immutable* state around, such as a program's configuration data. We could use the `State` monad for this purpose, but we could then find ourselves accidentally modifying data that should remain unchanged. No comments

Let's forget about monads for a moment and think about what a *function* with our desired characteristics ought to do. It should accept a value of some type `e` (for *environment*) that represents the data that we're passing in, and return a value of some other type `a` as its result. The overall type we want is `e -> a`. 3 comments

To turn this type into a convenient `Monad` instance, we'll wrap it in a `newtype`. No comments

```
-- file: ch15/SupplyInstance.hs
newtype Reader e a = R { runReader :: e -> a }
```

No comments

Making this into a `Monad` instance doesn't take much work. No comments

```
-- file: ch15/SupplyInstance.hs
instance Monad (Reader e) where
    return a = R $ \_ -> a
    m >>= k = R $ \r -> runReader (k (runReader m r)) r
```

5 comments

We can think of our value of type `e` as an *environment* in which we're evaluating some expression. The `return` action should have the same effect no matter what the environment is, so our version ignores its environment. 1 comment

Our definition of `(>>=)` is a little more complicated, but only because we have to make the environment—here the variable `r`—available both in the current computation and in the computation we're chaining into. No comments

How does a piece of code executing in this monad find out what's in its environment? It simply has to `ask`. 2 comments

```
-- file: ch15/SupplyInstance.hs
ask :: Reader e e
ask = R id
```

No comments

Within a given chain of actions, every invocation of `ask` will return the same value, since the value stored in the environment doesn't change. Our code is easy to test in **ghci**. No comments

```
ghci> runReader (ask >>= \x -> return (x * 3)) 2
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package random-1.0.0.0 ... linking ... done.
6
```

2 comments

The `Reader` monad is included in the standard `mtl` library, which is usually bundled with GHC. You can find it in the `Control.Monad.Reader` module. The motivation for this monad may initially seem a little thin, because it is most often useful in complicated code. We'll often need to access a piece of configuration information deep in the bowels of a program; passing that information in as a normal parameter would require a painful restructuring of our code. By hiding this information in our monad's plumbing, intermediate functions that don't care about the configuration information don't need to see it. No comments

The clearest motivation for the `Reader` monad will come in [Chapter 18, *Monad transformers*](), when we discuss combining several monads to build a new monad. There, we'll see how to gain finer control over state, so that our code can modify some values via the `State` monad, while other values remain immutable, courtesy of the `Reader` monad. 4 comments

A return to automated deriving

Now that we know about the `Reader` monad, let's use it to create an instance of our `MonadSupply` typeclass. To keep our example simple, we'll violate the spirit of `MonadSupply` here: our `next` action will always return the same value, instead of always returning a different value. 2 comments

It would be a bad idea to directly make the `Reader` type an instance of the `MonadSupply` class, because then *any* `Reader` could act as a `MonadSupply`. This would usually not make any sense. 1 comment

Instead, we create a `newtype` based on `Reader`. The `newtype` hides the fact that we're using `Reader` internally. We must now make our type an instance of both of the typeclasses we care about. With the `GeneralizedNewtypeDeriving` extension enabled, GHC will do most of the hard work for us. 4 comments

```
-- file: ch15/SupplyInstance.hs
newtype MySupply e a = MySupply { runMySupply :: Reader e a }
    deriving (Monad)

instance MonadSupply e (MySupply e) where
    next = MySupply $ do
             v <- ask
             return (Just v)

    -- more concise:
    -- next = MySupply (Just `liftM` ask)
```

4 comments

Notice that we must make our type an instance of `MonadSupply e`, not `MonadSupply`. If we omit the type variable, the compiler will complain. 4 comments

To try out our `MySupply` type, we'll first create a simple function that should work with any `MonadSupply` instance. 2 comments

```
-- file: ch15/SupplyInstance.hs
xy :: (Num s, MonadSupply s m) => m s
xy = do
  Just x <- next
  Just y <- next
  return (x * y)
```

4 comments

If we use this with our `Supply` monad and `randomsIO` function, we get a different answer every time, as we expect. No comments

```
ghci> (fst . runSupply xy) `fmap` randomsIO
-15697064270863081825448476392841917578
ghci> (fst . runSupply xy) `fmap` randomsIO
17182983444616834494257398042360119726
```

7 comments

Because our MySupply monad has two layers of newtype wrapping, we can make it easier to use by writing a custom execution function for it. No comments

```
-- file: ch15/SupplyInstance.hs
runMS :: MySupply i a -> i -> a
runMS = runReader . runMySupply
```

No comments

When we apply our xy action using this execution function, we get the same answer every time. Our code remains the same, but because we are executing it in a different implementation of MonadSupply, its behavior has changed. No comments

```
ghci> runMS xy 2
4
ghci> runMS xy 2
4
```

2 comments

Like our MonadSupply typeclass and Supply monad, almost all of the common Haskell monads are built with a split between interface and implementation. For example, the get and put functions that we introduced as “belonging to” the State monad are actually methods of the MonadState typeclass; the State type is an instance of this class. 1 comment

Similarly, the standard Reader monad is an instance of the MonadReader typeclass, which specifies the ask method. No comments

While the separation of interface and implementation that we've discussed above is appealing for its architectural cleanliness, it has important practical applications that will become clearer later. When we start combining monads in Chapter 18, *Monad transformers*, we will save a lot of effort through the use of GeneralizedNewtypeDeriving and typeclasses. 1 comment

Hiding the IO monad

The blessing and curse of the IO monad is that it is extremely powerful. If we believe that careful use of types helps us to avoid programming mistakes, then the IO monad should be a great source of unease. Because the IO monad imposes no restrictions on what we can do, it leaves us vulnerable to all kinds of accidents. No comments

How can we tame its power? Let's say that we would like to guarantee to ourselves that a piece of code can read and write files on the local filesystem, but that it will not access the network. We can't use the plain IO monad, because it won't restrict us. No comments

Using a newtype

Let's create a module that provides a small set of functionality for reading and writing files. No comments

```
-- file: ch15/HandleIO.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module HandleIO
    (
      HandleIO
    , Handle
    , IOMode(..)
    , runHandleIO
    , openFile
    , hClose
    , hPutStrLn
    ) where

import System.IO (Handle, IOMode(..))
import qualified System.IO
```

No comments

Our first approach to creating a restricted version of `IO` is to wrap it with a `newtype`. No comments

```
-- file: ch15/HandleIO.hs
newtype HandleIO a = HandleIO { runHandleIO :: IO a }
    deriving (Monad)
```

No comments

We do the by-now familiar trick of exporting the type constructor and the `runHandleIO` execution function from our module, but not the data constructor. This will prevent code running within the `HandleIO` monad from getting hold of the `IO` monad that it wraps. 1 comment

All that remains is for us to wrap each of the actions we want our monad to allow. This is a simple matter of wrapping each `IO` with a `HandleIO` data constructor. No comments

```
-- file: ch15/HandleIO.hs
openFile :: FilePath -> IOMode -> HandleIO Handle
openFile path mode = HandleIO (System.IO.openFile path mode)

hClose :: Handle -> HandleIO ()
hClose = HandleIO . System.IO.hClose

hPutStrLn :: Handle -> String -> HandleIO ()
hPutStrLn h s = HandleIO (System.IO.hPutStrLn h s)
```

1 comment

We can now use our restricted `HandleIO` monad to perform I/O. No comments

```
-- file: ch15/HandleIO.hs
safeHello :: FilePath -> HandleIO ()
safeHello path = do
  h <- openFile path WriteMode
  hPutStrLn h "hello world"
  hClose h
```

No comments

To run this action, we use `runHandleIO`. No comments

```
ghci> :load HandleIO
[1 of 1] Compiling HandleIO         ( HandleIO.hs, interpreted )
Ok, modules loaded: HandleIO.
ghci> runHandleIO (safeHello "hello_world_101.txt")
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> :m +System.Directory
ghci> removeFile "hello_world_101.txt"
```

3 comments

If we try to sequence an action that runs in the `HandleIO` monad with one that is not permitted, the type system forbids it. 1 comment

```
ghci> runHandleIO (safeHello "goodbye" >> removeFile "goodbye")

<interactive>:1:36:
    Couldn't match expected type `HandleIO a'
           against inferred type `IO ()'
    In the second argument of `(>>)', namely `removeFile "goodbye"'
    In the first argument of `runHandleIO', namely
        `(safeHello "goodbye" >> removeFile "goodbye")'
    In the expression:
        runHandleIO (safeHello "goodbye" >> removeFile "goodbye")
```

No comments

Designing for unexpected uses

There's one small, but significant, problem with our `HandleIO` monad: it doesn't take into account the possibility that we might occasionally need an escape hatch. If we define a monad like this, it is likely that we will occasionally need to perform an I/O action that isn't allowed for by the design of our monad. No comments

Our purpose in defining a monad like this is to make it easier for us to write solid code in the common case, not to make corner cases impossible. Let's thus give ourselves a way out. No comments

The `Control.Monad.Trans` module defines a “standard escape hatch”, the `MonadIO` typeclass. This defines a single function, `liftIO`, which lets us embed an `IO` action in another monad. No comments

```
ghci> :m +Control.Monad.Trans
ghci> :info MonadIO
class (Monad m) => MonadIO m where liftIO :: IO a -> m a
        -- Defined in Control.Monad.Trans
instance MonadIO IO -- Defined in Control.Monad.Trans
```

No comments

Our implementation of this typeclass is trivial: we just wrap `IO` with our data constructor. No comments

```
-- file: ch15/HandleIO.hs
import Control.Monad.Trans (MonadIO(..))

instance MonadIO HandleIO where
    liftIO = HandleIO
```

1 comment

With judicious use of `liftIO`, we can escape our shackles and invoke `IO` actions where necessary. No comments

```
-- file: ch15/HandleIO.hs
tidyHello :: FilePath -> HandleIO ()
tidyHello path = do
  safeHello path
  liftIO (removeFile path)
```

1 comment

Automatic derivation and MonadIO

We could have had the compiler automatically derive an instance of `MonadIO` for us by adding the type class to the `deriving` clause of `HandleIO`. In fact, in production code, this would be our usual strategy. We avoided that here simply to separate the presentation of the earlier material from that of `MonadIO`. No comments

Using typeclasses

The disadvantage of hiding `IO` in another monad is that we're still tied to a concrete implementation. If we want to swap `HandleIO` for some other monad, we must change the type of every action that uses `HandleIO`. No comments

As an alternative, we can create a typeclass that specifies the interface we want from a monad that manipulates files. No comments

```
-- file: ch15/MonadHandle.hs
{-# LANGUAGE FunctionalDependencies, MultiParamTypeClasses #-}

module MonadHandle (MonadHandle(..)) where

import System.IO (IOMode(..))

class Monad m => MonadHandle h m | m -> h where
    openFile :: FilePath -> IOMode -> m h
    hPutStr :: h -> String -> m ()
    hClose :: h -> m ()
    hGetContents :: h -> m String

    hPutStrLn :: h -> String -> m ()
    hPutStrLn h s = hPutStr h s >> hPutStr h "\n"
```

1 comment

Here, we've chosen to abstract away both the type of the monad and the type of a file handle. To satisfy the type checker, we've added a functional dependency: for any instance of `MonadHandle`, there is exactly one handle type that we can use. When we make the `IO` monad an instance of this class, we use a regular `Handle`. No comments

```
-- file: ch15/MonadHandleIO.hs
{-# LANGUAGE FunctionalDependencies, MultiParamTypeClasses #-}

import MonadHandle
import qualified System.IO

import System.IO (IOMode(..))
import Control.Monad.Trans (MonadIO(..), MonadTrans(..))
import System.Directory (removeFile)

import SafeHello

instance MonadHandle System.IO.Handle IO where
    openFile = System.IO.openFile
    hPutStr = System.IO.hPutStr
    hClose = System.IO.hClose
    hGetContents = System.IO.hGetContents
    hPutStrLn = System.IO.hPutStrLn
```

4 comments

Because any `MonadHandle` must also be a `Monad`, we can write code that manipulates files using normal `do` notation, without caring what monad it will finally execute in. No comments

```
-- file: ch15/SafeHello.hs
safeHello :: MonadHandle h m => FilePath -> m ()
safeHello path = do
  h <- openFile path WriteMode
  hPutStrLn h "hello world"
  hClose h
```

2 comments

Because we made `IO` an instance of this type class, we can execute this action from **ghci**. No comments

```
ghci> safeHello "hello to my fans in domestic surveillance"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> removeFile "hello to my fans in domestic surveillance"
```

2 comments

The beauty of the typeclass approach is that we can swap one underlying monad for another without touching much code, as most of our code doesn't know or care about the implementation. For instance, we could replace `IO` with a monad that compresses files as it writes them out. 2 comments

Defining a monad's interface through a typeclass has a further benefit. It lets other people hide our implementation in a `newtype` wrapper, and automatically derive instances of just the typeclasses they want to expose. 4 comments

Isolation and testing

In fact, because our `safeHello` function doesn't use the `IO` type, we can even use a monad that *can't* perform I/O. This allows us to test code that would normally have side effects in a completely pure, controlled environment. 1 comment

To do this, we will create a monad that doesn't perform I/O, but instead logs every file-related event for later processing. No comments

```
-- file: ch15/WriterIO.hs
data Event = Open FilePath IOMode
           | Put String String
           | Close String
           | GetContents String
             deriving (Show)
```

2 comments

Although we already developed a `Logger` type in the section called "Using a new monad: show your work!", here we'll use the standard, and more general, `Writer` monad. Like other `mtl` monads, the API provided by `Writer` is defined in a typeclass, in this case `MonadWriter`. Its most useful method is `tell`, which logs a value. 3 comments

```
ghci> :m +Control.Monad.Writer
ghci> :type tell
tell :: (MonadWriter w m) => w -> m ()
```

2 comments

The values we log can be of any `Monoid` type. Since the list type is a `Monoid`, we'll log to a list of `Event`.
3 comments

We could make `Writer [Event]` an instance of `MonadHandle`, but it's cheap, easy, and safer to make a special-purpose monad. No comments

```
-- file: ch15/WriterIO.hs
newtype WriterIO a = W { runW :: Writer [Event] a }
    deriving (Monad, MonadWriter [Event])
```

1 comment

Our execution function simply removes the `newtype` wrapper we added, then calls the normal `Writer` monad's execution function. No comments

```
-- file: ch15/WriterIO.hs
runWriterIO :: WriterIO a -> (a, [Event])
runWriterIO = runWriter . runW
```

No comments

When we try this code out in **ghci**, it gives us a log of the function's file activities. No comments

```
ghci> :load WriterIO
[1 of 3] Compiling MonadHandle      ( MonadHandle.hs, interpreted )
[2 of 3] Compiling SafeHello        ( SafeHello.hs, interpreted )
[3 of 3] Compiling WriterIO         ( WriterIO.hs, interpreted )
Ok, modules loaded: SafeHello, MonadHandle, WriterIO.
ghci> runWriterIO (safeHello "foo")
((),[Open "foo" WriteMode,Put "foo" "hello world",Put "foo" "\n",Close "foo"])
```

8 comments

The writer monad and lists

The writer monad uses the monoid's `mappend` function every time we use `tell`. Because `mappend` for lists is `(++)`, lists are not a good practical choice for use with `Writer`: repeated appends are expensive. We use lists above purely for simplicity. 1 comment

In production code, if you want to use the `Writer` monad and you need list-like behaviour, use a type with better append characteristics. One such type is the difference list, which we introduced in the section called “Taking advantage of functions as data”. You don't need to roll your own difference list implementation: a well tuned library is available for download from Hackage, the Haskell package database. Alternatively, you can use the `Seq` type from the `Data.Sequence` module, which we introduced in the section called “General purpose sequences”. 5 comments

Arbitrary I/O revisited

If we use the typeclass approach to restricting `IO`, we may still want to retain the ability to perform arbitrary I/O actions. We might try adding `MonadIO` as a constraint on our typeclass. No comments

```
-- file: ch15/MonadHandleIO.hs
class (MonadHandle h m, MonadIO m) => MonadHandleIO h m | m -> h

instance MonadHandleIO System.IO.Handle IO

tidierHello :: (MonadHandleIO h m) => FilePath -> m ()
tidierHello path = do
  safeHello path
  liftIO (removeFile path)
```

No comments

This approach has a problem, though: the added `MonadIO` constraint loses us the ability to test our code in a pure environment, because we can no longer tell whether a test might have damaging side effects. The alternative is to move this constraint from the typeclass, where it “infects” all functions, to only those functions that really need to perform I/O. No comments

```
-- file: ch15/MonadHandleIO.hs
tidyHello :: (MonadIO m, MonadHandle h m) => FilePath -> m ()
tidyHello path = do
  safeHello path
  liftIO (removeFile path)
```

No comments

We can use pure property tests on the functions that lack `MonadIO` constraints, and traditional unit tests on the rest. No comments

Unfortunately, we've substituted one problem for another: we can't invoke code with both `MonadIO` and `MonadHandle` constraints from code that has the `MonadHandle` constraint alone. If we find that somewhere deep in our `MonadHandle`-only code, we really need the `MonadIO` constraint, we must add it to all the code paths that lead to this point. 3 comments

Allowing arbitrary I/O is risky, and has a profound effect on how we develop and test our code. When we have to choose between being permissive on the one hand, and easier reasoning and testing on the other, we usually opt for the latter. No comments

Exercises

1. Using QuickCheck, write a test for an action in the `MonadHandle` monad, to see if it tries to write to a file handle that is not open. Try it out on `safeHello`. 6 comments
2. Write an action that tries to write to a file handle that it has closed. Does your test catch this bug? 1 comment
3. In a form-encoded string, the same key may appear several times, with or without values, e.g. `key&key=1&key=2`. What type might you use to represent the values associated with a key in this sort of string? Write a parser that correctly captures all of the information. 5 comments