This page will automatically redirect you. If it does not, you can click this link to get to the monad tutorial.

All About Monads

> A comprehensive guide to the theory and practice of monadic programming in Haskell<i>
> Version 1.1.0

This tutorial aims to explain the concept of a monad and its application to functional programming in a way that is easy to understand and useful to beginning and intermediate Haskell programmers. Familiarity with the Haskell language is assumed, but no prior experience with monads is required. The tutorial covers a lot of material and the later sections require a thorough understanding of the earlier material. Many code examples are provided along the way to demonstrate monadic programming. It is not advisable to attempt to absorb all of the material in a single reading. <p>

The tutorial is arranged in three parts. The first part provides a basic understanding of the role of monads in functional programming, how monads operate, and how they are declared and used in Haskell. The second part covers each standard monad in Haskell, giving the definition of the monad and discussing the use of the monad. The third part covers advanced material relating to monad transformers and real-world issues encountered when programming with monads.

You can also download the complete tutorial as a compressed tar archive or as a zip archive.

The best way to really understand monads is to experiment with monadic code. This tutorial provides many code examples which range from simple to moderately complex, and you are encouraged to study them, play with them, modify them, etc. Unfortunately, it is necessary that the early examples must be contrived to avoid issues with combining multiple monads. If you find yourself frustrated by this restriction while experimenting with the example code, look ahead at Part III to see how it is overcome.

Please send any comments, corrections, or suggestions to Jeff Newbern at jnewbern@yahoo.com

Contents

Part I - Understanding Monads

Part II - A Catalog of Standard Monads

Part III - Monads in the Real World

Appendices

Next: Introduction

Introduction

- What is a monad?
- Why should I make the effort to understand monads?

What is a monad?

A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.

It is useful to think of a monad as a strategy for combining computations into more complex computations. For example, you should be familiar with the `Maybe` type in Haskell:

```
data Maybe a = Nothing | Just a
```

which represents the type of computations which may fail to return a result. The `Maybe` type suggests a strategy for combining computations which return `Maybe` values: if a combined computation consists of one computation `B` that depends on the result of another computation `A`, then the combined computation should yield `Nothing` whenever either `A` or `B` yield `Nothing` and the combined computation should yield the result of `B` applied to the result of `A` when both computations succeed.

Other monads exist for building computations that perform I/O, have state, may return multiple results, etc. There are as many different type of monads as there are strategies for combining computations, but there are certain monads that are especially useful and are common enough that they are part of the standard Haskell 98 libraries. These monads are each described in Part II.

Why should I make the effort to understand monads?

The sheer number of different monad tutorials on the internet is a good indication of the difficulty many people have understanding the concept. This is due to the abstract nature of monads and to the fact that they are used in several different capacities, which can confuse the picture of exactly what a monad is and what it is good for.

In Haskell, monads play a central role in the I/O system. It is not essential to understand monads to do I/O in Haskell, but understanding the I/O monad will improve your code and extend your capabilities.

For the programmer, monads are useful tools for structuring functional programs. They have three properties that make them especially useful:

1. Modularity - They allow computations to be composed from simpler computations and separate the combination strategy from the actual computations being performed.
2. Flexibility - They allow functional programs to be much more adaptable than equivalent programs written without monads. This is because the monad distills the computational strategy into a single place instead of requiring it be distributed throughout the entire program.

3. Isolation - They can be used to create imperative-style computational structures which remain safely isolated from the main body of the functional program. This is useful for incorporating side-effects (such as I/O) and state (which violates referential transparency) into a pure functional language like Haskell.

Each of these features will be revisited later in the tutorial in the context of specific monads.

Prev: Table of Contents TOC: Contents Next: Meet the Monads

Haskell code examples

This appendix contains a list of all of the code examples supplied with the tutorial.

Example 1

This example is discussed in the section: Meet the monads.

The example code introduces the monad concept without using Haskell typeclasses. It shows how a monadic combinator can be used to simplify the construction of computations from sequences of computations which may not return a result.

Example 2

This example is discussed in the section: Doing it with class.

The example code builds on the first example, and shows how do-notation can be used with an instance of the `Monad` class (in this case, `Maybe` is the monad used).

Example 3

This example is discussed in the section: Monad support in Haskell.

The example code builds on the first two examples, and shows a somewhat atypical — but very powerful — use of the `foldM` function outside of a do-block.

Example 4

This example is discussed in the section: Monad support in Haskell.

The example code shows a more typical use of the `foldM` function within a do-block. It combines dictionary values read from different files into a single dictionary using the `foldM` function within the IO monad.

Example 5

This example is discussed in the section: Monad support in Haskell.

The example code shows the use of the `filterM` function within a do-block. It prints the subset of its arguments that specify directories and ignores non-directory arguments.

Example 6

This example is discussed in the section: Monad support in Haskell.

The example code shows the use of the `liftM` function within a do-block. It looks up a name in a list and uses a lifted String manipulation function to modify it within the Maybe monad.

Example 7

This example is discussed in the section: Monad support in Haskell.

The example code shows a higher-order application of `liftM2`. It folds lifted operations within the List monad to produce lists of all combinations of elements combined with the lifted operator.

Example 8

This example is discussed in the section: Monad support in Haskell.

The example code shows a higher-order application of `ap`. It folds `ap` through a list of `Maybe (a->a)` functions to process sequences of commands.

Example 9

This example is discussed in the section: Monad support in Haskell.

The example code shows the use of `msum` in the Maybe monad to select the first variable match in a stack of binding environments.

Example 10

This example is discussed in the section: Monad support in Haskell.

The example code shows the use of `guard` in the Maybe monad to select only the records from a list that satisfy a predicate (equivalent to `filter`).

Example 11

This example is discussed in the section: The Maybe monad.

The example code shows how to use the `Maybe` monad to build complex queries from simpler queries that may fail to return a result. The specific example used is looking up mail preferences for someone based on either their full name or a nickname.

Example 12

This example is discussed in the section: The Error monad.

The example code demonstrates the use of the `Either` type constructor as an `Error` monad with a custom error type. The example parses hexadecimal digits and uses the exception handling mechanism of the `Error` monad to provide informative error messages in the event of a parse failure.

Example 13

This example is discussed in the section: The List monad.

The example code uses the built-in list type constructor as a monad for non-deterministic computation. The example demonstrates parsing an ambiguous grammar consisting of integers, hex values, and words.

Example 14

This example is discussed in the section: The IO monad.

The example code implements a simple version of the standard Unix command "tr". The example demonstrates use of the IO monad including implicit `fail` calls due to pattern matching failures and the use of `catcherror`.

Example 15

This example is discussed in the section: The State monad.

The example code shows how the State monad can be used instead of explicitly passing state. The example uses the State monad to manage the random number generator state while building a compound data value requiring multiple calls to the random number generator.

Example 16

This example is discussed in the section: The Reader monad.

The example code shows how the Reader monad can be used to simplify computations involving a shared environment. The example uses the Reader monad to implement a simple template substitution system. The example code demonstrates the use of the Parsec monadic parser combinator library.

Example 17

This example is discussed in the section: The Writer monad.

The example code shows how the Writer monad can be used to implement logging. The example implements a very simple firewall simulator and uses the Writer monad to log the firewall activity.

Example 18

This example is discussed in the section: The Continuation monad.

The example code shows how the Continuation monad's escape continuations work. The example computes a complex transformation of a number.

Example 19

This example is discussed in the section: Combining monads the hard way.

The example code shows how the Continuation monad can be nested within the IO monad given a suitable computational structure. The example is a slight modification of example 18.

Example 20

This example is discussed in the section: Combining monads the hard way.

The example code shows how the Continuation monad and IO monad can be used simultaneously, but without using monad transformers. The example builds on examples 18 and 19.

Example 21

This example is discussed in the section: Monad transformers.

The example code shows how the transformer version of the Continuation monad can be used to create a combined monad for using continuations and doing I/O. The example builds on examples 18, 19 and 20.

Example 22

This example is discussed in the section: Standard monad transformers.

The example code shows how the transformer version of the Writer monad can be used to create a combined monad for logging and doing I/O. The example adds timestamps to the log entries of the firewall simulator from example 17.

Example 23

This example is discussed in the section: Standard monad transformers.

The example code shows how the transformer version of the Reader monad can be used to create a monad that combines a shared environment with I/O. The example converts the template system of example 16 to use files as templates.

Example 24

This example is discussed in the section: Standard monad transformers.

The example code uses the `StateT` transformer on the List monad to create a combined monad for doing non-deterministic stateful computations. The example uses the combined monad to solve a logic problem.

Example 25

This example is discussed in the section: An example with multiple monad transformers.

The example code uses the `StateT` and `WriterT` transformers on the List monad to create a combined monad for doing non-deterministic stateful computations with logging. The example uses the combined monad to solve the N-queens problem.

Example 26

This example is discussed in the section: Heavy lifting.

The example code demonstrates the use of the `lift` function and the necessity of managing its use in complex transformer stacks.

Prev: Apendix I - A physical analogy for monads

TOC: Contents

Meet the Monads

We will use the `Maybe` type constructor throughout this chapter, so you should familiarize yourself with the definition and usage of `Maybe` before continuing.

Type constructors

To understand monads in Haskell, you need to be comfortable dealing with type constructors. A *type constructor* is a parameterized type definition used with polymorphic types. By supplying a type constructor with one or more concrete types, you can construct a new concrete type in Haskell. In the definition of `Maybe`:

```
data Maybe a = Nothing | Just a
```

`Maybe` is a type constructor and `Nothing` and `Just` are data constructors. You can construct a data value by applying the `Just` data constructor to a value:

```
country = Just "China"
```

In the same way, you can construct a type by applying the `Maybe` type constructor to a type:

```
lookupAge :: DB -> String -> Maybe Int
```

Polymorphic types are like containers that are capable of holding values of many different types. So `Maybe Int` can be thought of as a `Maybe` container holding an `Int` value (or `Nothing`) and `Maybe String` would be a `Maybe` container holding a `String` value (or `Nothing`). In Haskell, we can also make the type of the container polymorphic, so we could write "`m a`" to represent a container of some type holding a value of some type!

We often use type variables with type constructors to describe abstract features of a computation. For example, the polymorphic type `Maybe a` is the type of all computations that may return a value or `Nothing`. In this way, we can talk about the properties of the container apart from any details of what the container might hold.

If you get messages about "kind errors" from the compiler when working with monads, it means that you are not using the type constructors correctly.

Maybe a monad

In Haskell a monad is represented as a type constructor (call it `m`), a function that builds values of that type (`a -> m a`), and a function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type (`m a -> (a -> m b) -> m b`). Note that the container is the same, but the type of the contents of the container can change. It is customary to call the monad type constructor "`m`" when discussing monads in general. The function that builds values of that type is traditionally called "`return`" and the third function is known as "bind" but is written ">>=". The signatures of the functions are:

```
-- the type of monad m
data m a = ...

-- return is a type constructor that creates monad instances
return :: a -> m a

-- bind is a function that combines a monad instance m a with a computation
-- that produces another monad instance m b from a's to produce a new
-- monad instance m b
(>>=) :: m a -> (a -> m b) -> m b
```

Roughly speaking, the monad type constructor defines a type of computation, the `return` function creates primitive values of that computation type and >>= combines computations of that type together to make more complex computations of that type. Using the container analogy, the type constructor `m` is a container that can hold different values. `m a` is a container holding a value of type `a`. The `return` function puts a value into a monad container. The >>= function takes the value from a monad container and passes it to a function to produce a monad container containing a new value, possibly of a different type. The >>= function is known as "bind" because it binds the value in a monad container to the first argument of a function. By adding logic to the binding function, a monad can implement a specific strategy for combining computations in the monad.

This will all become clearer after the example below, but if you feel particularly confused at this point you might try looking at this physical analogy of a monad before continuing.

An example

Suppose that we are writing a program to keep track of sheep cloning experiments. We would certainly want to know the genetic history of all of our sheep, so we would need `mother` and `father` functions. But since these are cloned sheep, they may not always have both a mother and a father!

We would represent the possibility of not having a mother or father using the `Maybe` type constructor in our Haskell code:

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

Then, defining functions to find grandparents is a little more complicated, because we have to handle the possibility of not having a parent:

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
                          Nothing -> Nothing
                          Just m  -> father m
```

and so on for the other grandparent combinations.

It gets even worse if we want to find great grandparents:

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m  -> case (father m) of
                                              Nothing -> Nothing
                                              Just gf -> father gf
```

Aside from being ugly, unclear, and difficult to maintain, this is just too much work. It is clear that a `Nothing` value at any point in the computation will cause `Nothing` to be the final result, and it would be much nicer to implement this notion once in a single place and remove all of the explicit `case` testing scattered all over the code. This will make the code easier to write, easier to read and easier to change. So good programming style would have us create a combinator that captures the behavior we want:

Code available in example1.hs

```
-- comb is a combinator for sequencing operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x

-- now we can use `comb` to build complicated sequences
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = (Just s) `comb` mother `comb` father `comb` father
```

The combinator is a huge success! The code is much cleaner and easier to write, understand and modify. Notice also that the `comb` function is entirely polymorphic — it is not specialized for `Sheep` in any way. In fact, *the combinator captures a general strategy for combining computations that may fail to return a value.* Thus, we can apply the same combinator to other computations that may fail to return a value, such as database queries or dictionary lookups.

The happy outcome is that common sense programming practice has led us to create a monad without even realizing it. The `Maybe` type constructor along with the `Just` function (acts like `return`) and our combinator (acts like >>=) together form a simple monad for building computations which may not return a value. All that remains to make this monad truly useful is to make it conform to the monad framework built into the Haskell language. That is the subject of the next chapter.

A list is also a monad

We have seen that the `Maybe` type constructor is a monad for building computations which may fail to return a value. You may be surprised to know that another common Haskell type constructor, `[]` (for building lists), is also a monad. The List monad allows us to build computations which can return 0, 1, or more values.

The `return` function for lists simply creates a singleton list (`return x = [x]`). The binding operation for lists creates a new list containing the results of applying the function to all of the values in the original list (`l >>= f = concatMap f l`).

One use of functions which return lists is to represent *ambiguous* computations — that is computations which may have 0, 1, or more allowed outcomes. In a computation composed from ambigous subcomputations, the ambiguity may compound, or it may eventually resolve into a single allowed outcome or no allowed outcome at all. During this process, the set of possible computational states is represented as a list. The List monad thus embodies a strategy for performing simultaneous computations along all allowed paths of an ambiguous computation.

Examples of this use of the List monad, and contrasting examples using the Maybe monad will be presented shortly. But first, we must see how useful monads are defined in Haskell.

Summary

We have seen that a monad is a type constructor, a function called `return`, and a combinator function called `bind` or `>>=`. These three elements work together to encapsulate a strategy for combining computations to produce more complex computations.

Using the `Maybe` type constructor, we saw how good programming practice led us to define a simple monad that could be used to build complex computations out of sequences of computations that could each fail to return a value. The resulting `Maybe` monad encapsulates a strategy for combining computations that may not return values. By codifying the strategy in a monad, we have achieved a degree of modularity and flexibility that is not present when the computations are combined in an ad hoc manner.

We have also seen that another common Haskell type constructor, `[]`, is a monad. The List monad encapsulates a strategy for combining computations that can return 0, 1, or multiple values.

Prev: Introduction | TOC: Contents | Next: Doing it with class

Doing it with class

Haskell type classes

The discussion in this chapter involves the Haskell type class system. If you are not familiar with type classes in Haskell, you should review them before continuing.

The Monad class

In Haskell, there is a standard `Monad` class that defines the names and signatures of the two monad functions `return` and `>>=`. It is not strictly necessary to make your monads instances of the `Monad` class, but it is a good idea. Haskell has special support for `Monad` instances built into the language and making your monads instances of the `Monad` class will allow you to use these features to write cleaner and more elegant code. Also, making your monads instances of the `Monad` class communicates important information to others who read the code and failing to do so can cause you to use confusing and non-standard function names. It's easy to do and it has many benefits, so just do it!

The standard `Monad` class definition in Haskell looks something like this:

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

Example continued

Continuing the previous example, we will now see how the `Maybe` type constructor fits into the Haskell monad framework as an instance of the `Monad` class.

Recall that our `Maybe` monad used the `Just` data constructor to fill the role of the monad `return` function and we built a simple combinator to fill the role of the monad `>>=` binding function. We can make its role as a monad explicit by declaring `Maybe` as an instance of the `Monad` class:

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Once we have defined `Maybe` as an instance of theMonad class, we can use the standard monad operators to build the complex computations:

```
-- we can use monadic operations to build complicated sequences
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = (return s) >>= mother >>= father

fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = (return s) >>= father >>= mother >>= mother
```

In Haskell, `Maybe` is defined as an instance of the `Monad` class in the standard prelude, so you don't need to do it yourself. The other monad we have seen so far, the list constructor, is also defined as an instance of the `Monad` class in the standard prelude.

When writing functions that work with monads, try to make use of the `Monad` class instead of using a specific monad instance. A function of the type

```
doSomething :: (Monad m) => a -> m b
```

is much more flexible than one of the type

```
doSomething :: a -> Maybe b
```

The former function can be used with many types of monads to get different behavior depending on the strategy embodied in the monad, whereas the latter function is restricted to the strategy of the `Maybe` monad.

Do notation

Using the standard monadic function names is good, but another advantage of membership in the `Monad` class is the Haskell support for "do" notation. Do notation is an expressive shorthand for building up monadic computations, similar to the way that list comprehensions are an expressive shorthand for building computations on lists. Any instance of the `Monad` class can be used in a do-block in Haskell.

In short, the do notation allows you to write monadic computations using a pseudo-imperative style with named variables. The result of a monadic computation can be "assigned" to a variable using a left arrow <- operator. Then using that variable in a subsequent monadic computation automatically performs the binding. The type of the expression to the right of the arrow is a monadic type `m a`. The expression to the left of the arrow is a pattern to be matched against the value *inside* the monad. `(x:xs)` would match against `Maybe [1,2,3]`, for example.

Here is a sample of do notation using the `Maybe` monad:

Code available in example2.hs

```
-- we can also use do-notation to build complicated sequences
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do m  <- mother s
                                  gf <- father m
                                  father gf
```

Compare this to `fathersMaternalGrandmother` written above without using do notation.

The do block shown above is written using the layout rule to define the extent of the block. Haskell also allows you to use braces and semicolons when defining a do block:

```
mothersPaternalGrandfather s = do { m <- mother s; gf <- father m; father gf }
```

Notice that do notation resembles an imperative programming language, in which a computation is built up from an explicit sequence of simpler computations. In this respect, monads offer the possibility to create imperative-style computations within a larger functional program. This theme will be expanded upon when we deal with side-effects and the I/O monad later.

Do notation is simply syntactic sugar. There is nothing that can be done using do notation that cannot be done using only the standard monadic operators. But do notation is cleaner and more convenient in some cases, especially when the sequence of monadic computations is long. You should understand both the standard monadic binding notation and do notation and be able to apply each where they are appropriate.

The actual translation from do notation to standard monadic operators is roughly that every expression matched to a pattern, `x <- expr1`, becomes

```
expr1 >>= \x ->
```

and every expression without a variable assignment, `expr2` becomes

```
expr2 >>= \_ ->
```

All do blocks must end with a monadic expression, and a let clause is allowed at the beginning of a do block (but let clauses in do blocks do not use the "in" keyword). The definition of `mothersPaternalGrandfather` above would be translated to:

```
mothersPaternalGrandfather s = mother s >>= \m ->
                               father m >>= \gf ->
                               father gf
```

It now becomes clear why the binding operator is so named. It is literally used to bind the value in the monad to the argument in the following lambda expression.

Summary

Haskell provides built-in support for monads. To take advantage of Haskell's monad support, you must declare the monad type constructor to be an instance of the `Monad` class and supply definitions of the `return` and >>= (pronounced "bind") functions for the monad.

A monad that is an instance of the `Monad` class can be used with do-notation, which is syntactic sugar that provides a simple, imperative-style notation for describing computations with monads.

Prev: Meet the Monads TOC: Contents Next: The monad laws

The monad laws

The tutorial up to now has avoided technical discussions, but there are a few technical points that must be made concerning monads. Monadic operations must obey a set of laws, known as "the monad axioms". These laws aren't enforced by the Haskell compiler, so it is up to the programmer to ensure that any `Monad` instances he declares obey they laws. Haskell's `Monad` class also includes some functions beyond the minimal complete definition that we have not seen yet. Finally, many monads obey additional laws beyond the standard monad laws, and there is an additional Haskell class to support these extended monads.

The three fundamental laws

The concept of a monad comes from a branch of mathematics called category theory. While it is not necessary to know category theory to create and use monads, we do need to obey a small bit of mathematical formalism. To create a monad, it is not enough just to declare a Haskell instance of the `Monad` class with the correct type signatures. To be a proper monad, the `return` and >>= functions must work together according to three laws:

1. `(return x) >>= f == f x`
2. `m >>= return == m`
3. `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

The first law requires that `return` is a left-identity with respect to >>=. The second law requires that `return` is a right-identity with respect to >>=. The third law is a kind of associativity law for >>=. Obeying the three laws ensures that the semantics of the do-notation using the monad will be consistent.

Any type constructor with return and bind operators that satisfy the three monad laws is a monad. In Haskell, the compiler does not check that the laws hold for every instance of the `Monad` class. It is up to the programmer to ensure that any `Monad` instance he creates satisfies the monad laws.

Failure IS an option

The definition of the `Monad` class given earlier showed only the minimal complete definition. The full definition of the `Monad` class actually includes two additional functions: `fail` and >>.

The default implementation of the `fail` function is:

```
fail s = error s
```

You do not need to change this for your monad unless you want to provide different behavior for failure or to incorporate failure into the computational strategy of your monad. The `Maybe` monad, for instance, defines `fail` as:

```
fail _ = Nothing
```

so that `fail` returns an instance of the `Maybe` monad with meaningful behavior when it is bound with other functions in the `Maybe` monad.

The `fail` function is not a required part of the mathematical definition of a monad, but it is included in the standard `Monad` class definition because of the role it plays in Haskell's do notation. The `fail` function is called whenever a pattern matching failure occurs in a do block:

```
fn :: Int -> Maybe [Int]
fn idx = do let l = [Just [1,2,3], Nothing, Just [], Just [7..20]]
            (x:xs) <- l!!idx   -- a pattern match failure will call "fail"
            return xs
```

So in the code above, `fn 0` has the value `Just [2,3]`, but `fn 1` and `fn 2` both have the value `Nothing`.

The >> function is a convenience operator that is used to bind a monadic computation that does not require input from the previous computation in the sequence. It is defined in terms of >>=:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= (\_ -> k)
```

No way out

You might have noticed that there is no way to get values out of a monad as defined in the standard `Monad` class. That is not an accident. Nothing prevents the monad author from allowing it using functions specific to the monad. For instance, values can be extracted from the `Maybe` monad by pattern matching on `Just x` or using the `fromJust` function.

By not requiring such a function, the Haskell `Monad` class allows the creation of *one-way* monads. One-way monads allow values to enter the monad through the `return` function (and sometimes the `fail` function) and they allow computations to be performed within the monad using the bind functions >>= and >>, but they do not allow values back out of the monad.

The `IO` monad is a familiar example of a one-way monad in Haskell. Because you can't escape from the `IO` monad, it is impossible to write a function that does a computation in the `IO` monad but whose result type does not include the `IO` type constructor. This means that *any* function whose result type does not contain the `IO` type constructor is guaranteed not to use the `IO` monad. Other monads, such as `List` and `Maybe`, do allow values out of the monad. So it is possible to write functions which use these monads internally but return non-monadic values.

The wonderful feature of a one-way monad is that it can support side-effects in its monadic operations but prevent them from destroying the functional properties of the non-monadic portions of the program.

Consider the simple issue of reading a character from the user. We cannot simply have a function `readChar :: Char`, because it needs to return a different character each time it is called, depending on the input from the user. It is an essential property of Haskell as a pure functional language that all functions return the same value when called twice with the same arguments. But it *is* ok to have an I/O function `getChar :: IO Char` in the `IO` monad, because it can only be used in a sequence within the one-way monad. There is no way to get rid of the `IO` type constructor in the signature of any function that uses it, so the `IO` type constructor acts as a kind of tag that identifies all functions that do I/O. Furthermore, such functions are only useful within the `IO` monad. So a one-way monad effectively creates an isolated computational domain in which the rules of a pure functional language can be relaxed. Functional computations can move into the domain, but dangerous side-effects and non-referentially-transparent functions cannot escape from it.

Another common pattern when defining monads is to represent monadic values as functions. Then when the value of a monadic computation is required, the resulting monad is "run" to provide the answer.

Zero and Plus

Beyond the three monad laws stated above, some monads obey additional laws. The monads have a special value `mzero` and an operator `mplus` that obey four additional laws:

1. `mzero >>= f == mzero`
2. `m >>= (\x -> mzero) == mzero`
3. ``mzero `mplus` m == m``
4. ``m `mplus` mzero == m``

It is easy to remember the laws for `mzero` and `mplus` if you associate `mzero` with 0, `mplus` with +, and >>= with × in ordinary arithmetic.

Monads which have a zero and a plus can be declared as instances of the `MonadPlus` class in Haskell:

```
class (Monad m) => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Continuing to use the `Maybe` monad as an example, we see that the `Maybe` monad is an instance of `MonadPlus`:

```
instance MonadPlus Maybe where
    mzero             = Nothing
    Nothing `mplus` x = x
    x `mplus` _       = x
```

This identifies `Nothing` as the zero value and says that adding two `Maybe` values together gives the first value that is not `Nothing`. If both input values are `Nothing`, then the result of `mplus` is also `Nothing`.

The List monad also has a zero and a plus. `mzero` is the empty list and `mplus` is the ++ operator.

The `mplus` operator is used to combine monadic values from separate computations into a single monadic value. Within the context of our sheep-cloning example, we could use `Maybe`'s `mplus` to define a function, `parent s = (mother s) ‘mplus‘ (father s)`, which would return a parent if there is one, and `Nothing` is the sheep has no parents at all. For a sheep with both parents, the function would return one or the other, depending on the exact definition of `mplus` in the `Maybe` monad.

Summary

Instances of the `Monad` class should conform to the so-called monad laws, which describe algabraic properties of monads. There are three of these laws which state that the `return` function is both a left and a right identity and that the binding operator is associative. Failure to satisfy these laws will result in monads that do not behave properly and may cause subtle problems when using do-notation.

In addition to the `return` and `>>=` functions, the `Monad` class defines another function, `fail`. The `fail` function is not a technical requirement for inclusion as a monad, but it is often useful in practice and it is included in the `Monad` class because it is used in Haskell's do-notation.

Some monads obey laws beyond the three basic monad laws. An important class of such monads are ones which have a notion of a zero element and a plus operator. Haskell provides a `MonadPlus` class for such monads which define the `mzero` value and the `mplus` operator.

Prev: Doing it with class TOC: Contents Next: Exercises

Exercises

This section contains a few simple exercises to hone the reader's monadic reasoning skills and to provide a solid comprehension of the function and use of the Maybe and List monads before looking at monadic programming in more depth. The exercises will build on the previous sheep-cloning example, with which the reader should already be familiar.

Exercise 1: Do notation

Rewrite the `maternalGrandfather`, `fathersMaternalGrandmother`, and `mothersPaternalGrandfather` functions in Example 2 using the monadic operators `return` and >>=, without using any do-notation syntactic sugar.

Click here to see the solution.

Exercise 2: Combining monadic values

Write functions `parent` and `grandparent` with signature `Sheep -> Maybe Sheep`. They should return one sheep selected from all sheep matching the description, or `Nothing` if there is no such sheep. Hint: the `mplus` operator is useful here.

Click here to see the solution.

Exercise 3: Using the List monad

Write functions `parent` and `grandparent` with signature `Sheep -> [Sheep]`. They should return all sheep matching the description, or the empty list if there is no such sheep. Hint: the `mplus` operator in the List monad is useful here. Also the `maybeToList` function in the `Maybe` module can be used to convert a value from the Maybe monad to the List monad.

Click here to see the solution.

Exercise 4: Using the Monad class constraint

Monads promote modularity and code reuse by encapsulating often-used computational strategies into single blocks of code that can be used to construct many different computations. Less obviously, monads also promote modularity by allowing you to vary the monad in which a computation is done to achieve different variations of the computation. This is achieved by writing functions which are polymorphic in the monad type constructor, using the `(Monad m) =>`, `(MonadPlus m) =>`, etc.

class constraints.

Write functions `parent` and `grandparent` with signature `(MonadPlus m) => Sheep -> m Sheep`. They should be useful in both the Maybe and List monads. How does the functions' behavior differ when used with the List monad versus the Maybe monad? If you need to review the use of type classes and class constraints in Haskell, look here.

Click here to see the solution.

Prev: The monad laws   TOC: Contents   Next: Monad support in Haskell

Solution to Exercise 1

A literal translation from do-notation would be:

```
maternalGrandfather s = mother s >>= \m ->
                        father m
```

but since the bound variable is used immediately in this function, the equivalent definition is more succinct:

Code available in exercise1.hs

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = mother s >>= father
```

The other two functions are defined similarly:

Code available in exercise1.hs

```
fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = father s >>= mother >>= mother

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = mother s >>= father >>= father
```

Return to exercises.

Solution to Exercise 2

The two choices for a parent are the mother and the father, suggesting a straightforward implementation of the `parent` function using the `mplus` operator:

Code available in exercise2.hs

```
parent :: Sheep -> Maybe Sheep
parent s = (mother s) `mplus` (father s)
```

In this case, the `mplus` operator is used to combine the values from the monadic functions `mother` and `father`.

The `grandparent` function is a little trickier. You might be tempted to simply write:

```
grandparent s = do p <- parent s
                   parent p
```

but this is not correct. To see why not, you must remember that the `mplus` operator in the Maybe monad returns its first argument unless it is `Nothing`, in which case it returns its second argument. Imagine what would happen if the sheep had a mother without parents but a father with parents. Then `p` would be the mother and `parent p` would be `Nothing`, even though there are grandparents on the father's side.

The problem is that there is no back-tracking facility in the Maybe monad's strategy for `mplus`. We can get around this by carefully defining the `grandparent` function so that no backtracking is required:

Code available in exercise2.hs

```
grandparent :: Sheep -> Maybe Sheep
grandparent s = (mother s >>= parent) `mplus` (father s >>= parent)
```

Return to exercises.

Solution to Exercise 3

The definition of `parent` in the List monad is very similar to the definition in the Maybe monad. The only difference is the need to convert the values returned by `mother` and `father` from the Maybe monad to the List monad using the `Maybe.maybeToList` function.

Code available in exercise3.hs

```
parent :: Sheep -> [Sheep]
parent s = (maybeToList (mother s)) `mplus` (maybeToList (father s))
```

Unlike our definition of `grandparent` in the Maybe monad, we can use the simpler form in the List monad since the List monad does implement a backtracking strategy:

Code available in exercise3.hs

```
grandparent :: Sheep -> [Sheep]
grandparent s = do p <- parent s
                   parent p
```

Return to exercises.

Solution to Exercise 4

Writing code which can be used in multiple monads requires careful thought about the definitions of each of the monads in which the computation is to be performed. The functions must be implemented using the subset of features common to all of the required monads.

A useful function in this instance is one to convert a value in the Maybe monad into any instance of the `MonadPlus` class:

Code available in exercise4.hs

```
-- convert a Maybe value into another monad
maybeToMonad :: (MonadPlus m) => Maybe a -> m a
maybeToMonad Nothing  = mzero
maybeToMonad (Just s) = return s
```

Note that this function acts like `id` when used in the Maybe monad. Using this function, we can define a version of `parent` similar to the definition we used in the List monad:

Code available in exercise4.hs

```
parent :: (MonadPlus m) => Sheep -> m Sheep
parent s = (maybeToMonad (mother s)) `mplus` (maybeToMonad (father s))
```

Because of the non-backtracking limitation of the Maybe monad, our definition of the `grandparent` function follows the definition we used with the Maybe monad, but with the addition of the calls to `maybeToMonad` necessary for use in the List monad:

Code available in exercise4.hs

```
grandparent :: (MonadPlus m) => Sheep -> m Sheep
grandparent s = (maybeToMonad (mother s) >>= parent) `mplus`
                (maybeToMonad (father s) >>= parent)
```

Return to exercises.

Monad support in Haskell

- In the standard prelude
- In the Monad module
- Summary

Haskell's built in support for monads is split among the standard prelude, which exports the most common monad functions, and the Monad module, which contains less-commonly used monad functions. The individual monad types are each in their own libraries and are the subject of Part II of this tutorial.

In the standard prelude

The Haskell 98 standard prelude includes the definition of the `Monad` class as well as a few auxilliary functions for working with monadic data types.

The `Monad` class

We have seen the `Monad` class before:

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

The sequencing functions

The `sequence` function takes a list of monadic computations, executes each one in turn and returns a list of the results. If any of the computations fail, then the whole function fails:

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
             where mcons p q = p >>= \x -> q >>= \y -> return (x:y)
```

The `sequence_` function (notice the underscore) has the same behavior as `sequence` but does not return a list of results. It is useful when only the side-effects of the monadic computations are important.

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

The mapping functions

The mapM function maps a monadic computation over a list of values and returns a list of the results. It is defined in terms of the list map function and the sequence function above:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
```

There is also a version with an underscore, mapM_ which is defined using sequence_. mapM_ operates the same as mapM, but it doesn't return the list of values. It is useful when only the side-effects of the monadic computation are important.

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

As a simple example of the use the mapping functions, a putString function for the IO monad could be defined as:

```
putString :: [Char] -> IO ()
putString s = mapM_ putChar s
```

mapM can be used within a do block in a manner similar to the way the map function is normally used on lists. This is a common pattern with monads — a version of a function for use within a monad (i.e., intended for binding) will have a signature similar to the non-monadic version but the function outputs will be within the monad:

```
-- compare the non-monadic and monadic signatures
map  ::            (a -> b)   -> [a] -> [b]
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

The reverse binder function (=<<)

The prelude also defines a binding function that takes it arguments in the opposite order to the standard binding function. Since the standard binding function is called ">>=", the reverse binding function is called "=<<". It is useful in circumstances where the binding operator is used as a higher-order term and it is more convenient to have the arguments in the reversed order. Its definition is simply:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
f =<< x = x >>= f
```

In the Monad module

The Monad module in the standard Haskell 98 libraries exports a number of facilities for more advanced monadic operations. To access these facilities, simply import Monad in your Haskell program.

Not all of the function in the `Monad` module are discussed here, but you are encouraged to explore the module for yourself when you feel you are ready to see some of the more esoteric monad functions.

The `MonadPlus` class

The `Monad` module defines the `MonadPlus` class for monads with a zero element and a plus operator:

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Monadic versions of list functions

Several functions are provided which generalize standard list-processing functions to monads. The `mapM` functions are exported in the standard prelude and were described above.

`foldM` is a monadic version of `foldl` in which monadic computations built from a list are bound left-to-right. The definition is:

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a []     = return a
foldM f a (x:xs) = f a x >>= \y -> foldM f y xs
```

but it is easier to understand the operation of `foldM` if you consider its effect in terms of a do block:

```
-- this is not valid Haskell code, it is just for illustration
foldM f a1 [x1,x2,...,xn] = do a2 <- f a1 x1
                               a3 <- f a2 x2
                               ...
                               f an xn
```

Right-to-left binding is achieved by reversing the input list before calling `foldM`.

We can use `foldM` to create a more poweful query function in our sheep cloning example:

Code available in example3.hs

```
-- traceFamily is a generic function to find an ancestor
traceFamily :: Sheep -> [ (Sheep -> Maybe Sheep) ] -> Maybe Sheep
traceFamily s l = foldM getParent s l
  where getParent s f = f s

-- we can define complex queries using traceFamily in an easy, clear way
mothersPaternalGrandfather s = traceFamily s [mother, father, father]
paternalGrandmother s = traceFamily s [father, mother]
```

The `traceFamily` function uses `foldM` to create a simple way to trace back in the family tree to any depth and in any pattern. In fact, it is probably clearer to write "`traceFamily s [father, mother]`" than it is to use the `paternalGrandmother` function!

A more typical use of `foldM` is within a do block:

Code available in example4.hs

```
-- a Dict is just a finite map from strings to strings
type Dict = FiniteMap String String

-- this an auxilliary function used with foldl
addEntry :: Dict -> Entry -> Dict
addEntry d e = addToFM d (key e) (value e)

-- this is an auxiliiary function used with foldM inside the IO monad
addDataFromFile :: Dict -> Handle -> IO Dict
addDataFromFile dict hdl = do contents <- hGetContents hdl
                              entries  <- return (map read (lines contents))
                              return (foldl (addEntry) dict entries)

-- this program builds a dictionary from the entries in all files named on the
-- command line and then prints it out as an association list
main :: IO ()
main = do files   <- getArgs
          handles <- mapM openForReading files
          dict    <- foldM addDataFromFile emptyFM handles
          print (fmToList dict)
```

The `filterM` function works like the list `filter` function inside of a monad. It takes a predicate function which returns a Boolean value in the monad and a list of values. It returns, inside the monad, a list of those values for which the predicate was True.

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then (x:ys) else ys)
```

Here is an example showing how `filterM` can be used within the `IO` monad to select only the directories from a list:

Code available in example5.hs

```
import Monad
import Directory
import System

-- NOTE: doesDirectoryExist has type FilePath -> IO Bool

-- this program prints only the directories named on the command line
main :: IO ()
main = do names <- getArgs
          dirs  <- filterM doesDirectoryExist names
          mapM_ putStrLn dirs
```

`zipWithM` is a monadic version of the `zipWith` function on lists. `zipWithM_` behaves the same but discards the output of the function. It is useful when only the side-effects of the monadic computation matter.

```
zipWithM ::(Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys = sequence (zipWith f xs ys)

zipWithM_ ::(Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys = sequence_ (zipWith f xs ys)
```

Conditional monadic computations

There are two functions provided for conditionally executing monadic computations. The `when` function takes a boolean argument and a monadic computation with unit "()" type and performs the computation only when the boolean argument is `True`. The `unless` function does the same, except that it performs the computation *unless* the boolean argument is `True`.

```
when :: (Monad m) => Bool -> m () -> m ()
when p s = if p then s else return ()

unless :: (Monad m) => Bool -> m () -> m ()
unless p s = when (not p) s
```

`ap` and the lifting functions

Lifting is a monadic operation that converts a non-monadic function into an equivalent function that operates on monadic values. We say that a function is "lifted into the monad" by the lifting operators. A lifted function is useful for operating on monad values outside of a do block and can also allow for cleaner code within a do block.

The simplest lifting operator is `liftM`, which lifts a function of a single argument into a monad.

```
liftM :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f = \a -> do { a' <- a; return (f a') }
```

Lifting operators are also provided for functions with more arguments. `liftM2` lifts functions of two arguments:

```
liftM2 :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f = \a b -> do { a' <- a; b' <- b; return (f a' b') }
```

The same pattern is applied to give the definitions to lift functions of more arguments. Functions up to `liftM5` are defined in the `Monad` module.

To see how the lifting operators allow more concise code, consider a computation in the `Maybe` monad in which you want to use a function `swapNames::String -> String`. You could do:

```
getName :: String -> Maybe String
getName name = do let db = [("John", "Smith, John"), ("Mike", "Caine, Michael")]
                  tempName <- lookup name db
                  return (swapNames tempName)
```

But making use of the `liftM` function, we can use `liftM swapNames` as a function of type `Maybe String -> Maybe String`:

Code available in example6.hs

```
getName :: String -> Maybe String
getName name = do let db = [("John", "Smith, John"), ("Mike", "Caine, Michael")]
                  liftM swapNames (lookup name db)
```

The difference is even greater when lifting functions with more arguments.

The lifting functions also enable very concise constructions using higher-order functions. To understand this example code, you might need to review the definition of the monad functions for the List monad (particularly >>=). Imagine how you might implement this function without lifting the operator:

Code available in example7.hs

```
-- allCombinations returns a list containing the result of
-- folding the binary operator through all combinations
-- of elements of the given lists
-- For example, allCombinations (+) [[0,1],[1,2,3]] would be
--   [0+1,0+2,0+3,1+1,1+2,1+3], or [1,2,3,2,3,4]
-- and allCombinations (*) [[0,1],[1,2],[3,5]] would be
--   [0*1*3,0*1*5,0*2*3,0*2*5,1*1*3,1*1*5,1*2*3,1*2*5], or [0,0,0,0,3,5,6,10]
allCombinations :: (a -> a -> a) -> [[a]] -> [a]
allCombinations fn []     = []
allCombinations fn (l:ls) = foldl (liftM2 fn) l ls
```

There is a related function called `ap` that is sometimes more convenient to use than the lifting functions. `ap` is simply the function application operator (`$`) lifted into the monad:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap = liftM2 ($)
```

Note that `liftM2 f x y` is equivalent to `return f 'ap' x 'ap' y`, and so on for functions of more arguments. `ap` is useful when working with higher-order functions and monads.

The effect of `ap` depends on the strategy of the monad in which it is used. So for example `[(*2),(+3)] 'ap' [0,1,2]` is equal to `[0,2,4,3,4,5]` and `(Just (*2)) 'ap' (Just 3)` is `Just 6`. Here is a simple example that shows how `ap` can be useful when doing higher-order computations:

Code available in example8.hs

```
-- lookup the commands and fold ap into the command list to
-- compute a result.
main :: IO ()
main = do let fns  = [("double",(2*)),      ("halve",(`div`2)),
                      ("square",(\x->x*x)), ("negate", negate),
                      ("incr",(+1)),        ("decr",(+(-1)))
                     ]
          args <- getArgs
          let val  = read (args!!0)
              cmds = map ((flip lookup) fns) (words (args!!1))
          print $ foldl (flip ap) (Just val) cmds
```

Functions for use with `MonadPlus`

There are two functions in the `Monad` module that are used with monads that have a zero and a plus. The first function is `msum`, which is analogous to the `sum` function on lists of integers. `msum` operates on lists of monadic values and folds the `mplus` operator into the list using the `mzero` element as the initial value:

```
msum :: MonadPlus m => [m a] -> m a
msum xs = foldr mplus mzero xs
```

In the List monad, `msum` is equivalent to `concat`. In the `Maybe` monad, `msum` returns the first non-`Nothing` value from a list. Likewise, the behavior in other monads will depend on the exact nature of their `mzero` and `mplus` definitions.

`msum` allows many recursive functions and folds to be expressed more concisely. In the `Maybe` monad, for example, we can write:

Code available in example9.hs

```
type Variable = String
type Value = String
type EnvironmentStack = [[(Variable,Value)]]

-- lookupVar retrieves a variable's value from the environment stack
-- It uses msum in the Maybe monad to return the first non-Nothing value.
lookupVar :: Variable -> EnvironmentStack -> Maybe Value
lookupVar var stack = msum $ map (lookup var) stack
```

instead of:

```
lookupVar :: Variable -> EnvironmentStack -> Maybe Value
lookupVar var []     = Nothing
lookupVar var (e:es) = let val = lookup var e
                       in maybe (lookupVar var es) Just val
```

The second function for use with monads with a zero and a plus is the `guard` function:

```
guard :: MonadPlus m => Bool -> m ()
guard p = if p then return () else mzero
```

The trick to understanding this function is to recall the law for monads with zero and plus that states `mzero >>= f == mzero`. So, placing a `guard` function in a sequence of monadic operations will force any execution in which the guard is `False` to be `mzero`. This is similar to the way that guard predicates in a list comprehension cause values that fail the predicate to become `[]`.

Here is an example demonstrating the use of the `guard` function in the `Maybe` monad.

Code available in example10.hs

```
data Record = Rec {name::String, age::Int} deriving Show
type DB = [Record]

-- getYoungerThan returns all records for people younger than a specified age.
-- It uses the guard function to eliminate records for ages at or over the limit.
-- This is just for demonstration purposes.  In real life, it would be
-- clearer to simply use filter.  When the filter criteria are more complex,
-- guard becomes more useful.
getYoungerThan :: Int -> DB -> [Record]
getYoungerThan limit db = mapMaybe (\r -> do { guard (age r < limit); return r }) db
```

Summary

Haskell provides a number of functions which are useful for working with monads in the standard libraries. The `Monad` class and most common monad functions are in the standard prelude. The `MonadPlus` class and less commonly-used (but still very useful!) functions are defined in the `Monad` module. Many other types in the Haskell libraries are declared as instances of `Monad` and `MonadPlus` in their respective modules.

Prev: Exercises TOC: Contents Next: Part II - Introduction

Introduction

The monads covered in Part II include a mixture of standard Haskell types that are monads as well as monad classes from Andy Gill's Monad Template Library. The Monad Template Library is included in the Glasgow Haskell Compiler's hierarchical libraries under Control.Monad

Some of the documentation for these monads comes from the excellent Haskell Wiki. In addition to the monads covered here, monads appear many other places in Haskell, such as the Parsec monadic combinator parsing library. These monads are beyond the scope of this reference, but they are thoroughly documented on their own. You can get a taste of the Parsec library by looking in the source code for example 16.

Monad	Type of computation	Combination strategy for >>=
Identity	*N/A — Used with monad transformers*	The bound function is applied to the input value.
Maybe	Computations which may not return a result	`Nothing` input gives `Nothing` output `Just x` input uses `x` as input to the bound function.
Error	Computations which can fail or throw exceptions	Failure records information describing the failure. Binding passes failure information on without executing the bound function, or uses successful values as input to the bound function.
[] (List)	Non-deterministic computations which can return multiple possible results	Maps the bound function onto the input list and concatenates the resulting lists to get a list of all possible results from all possible inputs.
IO	Computations which perform I/O	Sequential execution of I/O actions in the order of binding.
State	Computations which maintain state	The bound function is applied to the input value to produce a state transition function which is applied to the input state.
Reader	Computations which read from a shared environment	The bound function is applied to the value of the input using the same environment.
Writer	Computations which write data in addition to computing values	Written data is maintained separately from values. The bound function is applied to the input value and anything it writes is appended to the write data stream.
Cont	Computations which can be interrupted and restarted	The bound function is inserted into the continuation chain.

The Identity monad

- Overview
- Motivation
- Definition
- Example

Overview

Computation type:	Simple function application
Binding strategy:	The bound function is applied to the input value. `Identity x >>= f == Identity (f x)`
Useful for:	Monads can be derived from monad transformers applied to the Identity monad.
Zero and plus:	None.
Example type:	Identity a

Motivation

The Identity monad is a monad that does not embody any computational strategy. It simply applies the bound function to its input without any modification. Computationally, there is no reason to use the Identity monad instead of the much simpler act of simply applying functions to their arguments. The purpose of the Identity monad is its fundamental role in the theory of monad transformers (covered in Part III). Any monad transformer applied to the Identity monad yields a non-transformer version of that monad.

Definition

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a           = Identity a   -- i.e. return = id
    (Identity x) >>= f = f x          -- i.e. x >>= f = f x
```

The `runIdentity` label is used in the type definition because it follows a style of monad definition that explicitly represents monad values as computations. In this style, a monadic computation is built up using the monadic operators and then the value of the computation is extracted using the `run******` function. Because the Identity monad does not do any computation, its definition is trivial. For a better example of this style of monad, see the State monad.

Example

A typical use of the Identity monad is to derive a monad from a monad transformer.

```
-- derive the State monad using the StateT monad transformer
type State s a = StateT s Identity a
```

Prev: Part II - Introduction TOC: Contents Next: The Maybe monad

The Maybe monad

Overview

Computation type:	Computations which may return `Nothing`
Binding strategy:	`Nothing` values bypass the bound function, other values are used as inputs to the bound function.
Useful for:	Building computations from sequences of functions that may return `Nothing`. Complex database queries or dictionary lookups are good examples.
Zero and plus:	`Nothing` is the zero. The plus operation returns the first non-`Nothing` value or `Nothing` is both inputs are `Nothing`.
Example type:	Maybe a

Motivation

The Maybe monad embodies the strategy of combining a chain of computations that may each return `Nothing` by ending the chain early if any step produces `Nothing` as output. It is useful when a computation entails a sequence of steps that depend on one another, and in which some steps may fail to return a value.

If you ever find yourself writing code like this:

```
case ... of
  Nothing -> Nothing
  Just x  -> case ... of
               Nothing -> Nothing
               Just y  -> ...
```

you should consider using the monadic properties of `Maybe` to improve the code.

Definition

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return         = Just
    fail           = Nothing
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x

instance MonadPlus Maybe where
    mzero             = Nothing
    Nothing `mplus` x = x
    x `mplus` _       = x
```

Example

A common example is in combining dictionary lookups. Given a dictionary that maps full names to email addresses, another that maps nicknames to email addresses, and a third that maps email addresses to email preferences, you could create a function that finds a person's email preferences based on either a full name or a nickname.

Code available in example11.hs

```
data MailPref = HTML | Plain
data MailSystem = ...

getMailPrefs :: MailSystem -> String -> Maybe MailPref
getMailPrefs sys name =
  do let nameDB = fullNameDB sys
         nickDB = nickNameDB sys
         prefDB = prefsDB sys
  addr <- (lookup name nameDB) `mplus` (lookup name nickDB)
  lookup addr prefDB
```

Prev: The Identity monad   TOC: Contents   Next: The Error monad

The Error monad

- Overview
- Motivation
- Definition
- Example

Overview

Computation type:	Computations which may fail or throw exceptions
Binding strategy:	Failure records information about the cause/location of the failure. Failure values bypass the bound function, other values are used as inputs to the bound function.
Useful for:	Building computations from sequences of functions that may fail or using exception handling to structure error handling.
Zero and plus:	Zero is represented by an empty error and the plus operation executes its second argument if the first fails.
Example type:	Either String a

Motivation

The Error monad (also called the Exception monad) embodies the strategy of combining computations that can throw exceptions by bypassing bound functions from the point an exception is thrown to the point that it is handled.

The `MonadError` class is parameterized over the type of error information and the monad type constructor. It is common to use `Either String` as the monad type constructor for an error monad in which error descriptions take the form of strings. In that case and many other common cases the resulting monad is already defined as an instance of the `MonadError` class. You can also define your own error type and/or use a monad type constructor other than `Either String` or `Either IOError`. In these cases you will have to explicitly define instances of the `Error` and/or `MonadError` classes.

Definition

The definition of the `MonadError` class below uses multi-parameter type classes and funDeps, which are language extensions not found in standard Haskell 98. You don’t need to understand them to take advantage of the `MonadError` class.

```
class Error a where
    noMsg :: a
    strMsg :: String -> a

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

`throwError` is used within a monadic computation to begin exception processing. `catchError` provides a handler function to handle previous errors and return to normal execution. A common idiom is:

```
do { action1; action2; action3 } `catchError` handler
```

where the `action` functions can call `throwError`. Note that `handler` and the do-block must have the same return type.

The definition of the `Either e` type constructor as an instance of the `MonadError` class is straightforward. Following convention, `Left` is used for error values and `Right` is used for non-error (right) values.

```
instance MonadError (Either e) where
    throwError = Left
    (Left e) `catchError` handler = handler e
    a        `catchError` _       = a
```

Example

Here is an example that demonstrates the use of a custom `Error` data type with the `ErrorMonad`'s `throwError` and `catchError` exception mechanism. The example attempts to parse hexadecimal numbers and throws an exception if an invalid character is encountered. We use a custom `Error` data type to record the location of the parse error. The exception is caught by a calling function and handled by printing an informative error message.

Code available in example12.hs

```
-- This is the type of our parse error representation.
data ParseError = Err {location::Int, reason::String}

-- We make it an instance of the Error class
instance Error ParseError where
  noMsg    = Err 0 "Parse Error"
  strMsg s = Err 0 s

-- For our monad type constructor, we use Either ParseError
-- which represents failure using Left ParseError or a
-- successful result of type a using Right a.
type ParseMonad = Either ParseError

-- parseHexDigit attempts to convert a single hex digit into
-- an Integer in the ParseMonad monad and throws an error on an
-- invalid character
parseHexDigit :: Char -> Int -> ParseMonad Integer
parseHexDigit c idx = if isHexDigit c then
                        return (toInteger (digitToInt c))
                      else
                        throwError (Err idx ("Invalid character '" ++ [c] ++ "'"))

-- parseHex parses a string containing a hexadecimal number into
-- an Integer in the ParseMonad monad.  A parse error from parseHexDigit
-- will cause an exceptional return from parseHex.
parseHex :: String -> ParseMonad Integer
parseHex s = parseHex' s 0 1
  where parseHex' []      val _   = return val
        parseHex' (c:cs)  val idx = do d <- parseHexDigit c idx
                                       parseHex' cs ((val * 16) + d) (idx + 1)

-- toString converts an Integer into a String in the ParseMonad monad
toString :: Integer -> ParseMonad String
toString n = return $ show n

-- convert takes a String containing a hexadecimal representation of
-- a number to a String containing a decimal representation of that
-- number.  A parse error on the input String will generate a
-- descriptive error message as the output String.
convert :: String -> String
convert s = let (Right str) = do {n <- parseHex s; toString n} `catchError` printError
            in str
  where printError e = return $ "At index " ++ (show (location e)) ++ ":" ++ (reason e)
```

Prev: The Maybe monad TOC: Contents Next: The List monad

The List monad

- Overview
- Motivation
- Definition
- Example

Overview

Computation type:	Computations which may return 0, 1, or more possible results.
Binding strategy:	The bound function is applied to all possible values in the input list and the resulting lists are concatenated to produce a list of all possible results.
Useful for:	Building computations from sequences of non-deterministic operations. Parsing ambiguous grammars is a common example.
Zero and plus:	`[]` is the zero and ++ is the plus operation.
Example type:	`[a]`

Motivation

The List monad embodies the strategy of combining a chain of non-deterministic computations by applying the operations to all possible values at each step. It is useful when computations must deal with ambiguity. In that case it allows all possibilities to be explored until the ambiguity is resolved.

Definition

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
    fail s   = []

instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

Example

The canonical example of using the List monad is for parsing ambiguous grammars. The example below shows just a small example of parsing data into hex values, decimal values and words containing only alpha-numeric characters. Note that hexadecimal digits overlap both decimal digits and alphanumeric characters, leading to an ambiguous grammar. "dead" is both a valid hex value and a word, for example, and "10" is both a decimal value of 10 and a hex value of 16.

Code available in example13.hs

```
-- we can parse three different types of terms
data Parsed = Digit Integer | Hex Integer | Word String deriving Show

-- attempts to add a character to the parsed representation of a hex digit
parseHexDigit :: Parsed -> Char -> [Parsed]
parseHexDigit (Hex n) c = if isHexDigit c then
                            return (Hex ((n*16) + (toInteger (digitToInt c))))
                          else
                            mzero
parseHexDigit _       _ = mzero

-- attempts to add a character to the parsed representation of a decimal digit
parseDigit :: Parsed -> Char -> [Parsed]
parseDigit (Digit n) c = if isDigit c then
                           return (Digit ((n*10) + (toInteger (digitToInt c))))
                         else
                           mzero
parseDigit _         _ = mzero

-- attempts to add a character to the parsed representation of a word
parseWord :: Parsed -> Char -> [Parsed]
parseWord (Word s) c = if isAlpha c then
                         return (Word (s ++ [c]))
                       else
                         mzero
parseWord _        _ = mzero

-- tries to parse the digit as a hex value, a decimal value and a word
-- the result is a list of possible parses
parse :: Parsed -> Char -> [Parsed]
parse p c = (parseHexDigit p c) `mplus` (parseDigit p c) `mplus` (parseWord p c)

-- parse an entire String and return a list of the possible parsed values
parseArg :: String -> [Parsed]
parseArg s = do init <- (return (Hex 0)) `mplus` (return (Digit 0)) `mplus` (return (Word ""))
                foldM parse init s
```

Prev: The Error monad TOC: Contents Next: The IO monad

The IO monad

- Overview
- Motivation
- Definition
- Example

Overview

Computation type:	Computations which perform I/O
Binding strategy:	I/O actions are executed in the order in which they are bound. Failures throw I/O errors which can be caught and handled.
Useful for:	Performing I/O within a Haskell program.
Zero and plus:	None.
Example type:	IO a

Motivation

Input/Output is incompatible with a pure functional language because it is not referentially transparent and side-effect free. The IO monad solves this problem by confining computations that perform I/O within the IO monad.

Definition

The definition of the IO monad is platform-specific. No data constructors are exported and no functions are provided to remove data from the IO monad. This makes the IO monad a one-way monad and is essential to ensuring safety of functional programs by isolating side-effects and non-referentially transparent actions within the imperative-style computations of the IO monad.

Throughout this tutorial, we have referred to monadic values as computations. However, values in the IO monad are often called *I/O actions* and we will use that terminology here.

In Haskell, the top-level `main` function must have type `IO ()`, so that programs are typically structured at the top level as an imperative-style sequence of I/O actions and calls to functional-style code. The functions exported from the `IO` module do not perform I/O themselves. They return I/O actions, which describe an I/O operation to be performed. The I/O actions are combined within the IO monad (in a purely functional manner) to create more complex I/O actions, resulting in the final I/O action that is the `main` value of the program.

The standard prelude and the IO module define many functions that can be used within the IO monad and any Haskell programmer will undoubtedly be familiar with some of them. This tutorial will only discuss the monadic aspects of the IO monad, not the full range of functions available to perform I/O.

The IO type constructor is a member of the Monad class and the MonadError class, where errors are of the type IOError. fail is defined to throw an error built from the string argument. Within the IO monad you can use the exception mechanisms from the Control.Monad.Error module in the Monad Template Library if you import the module. The same mechanisms have alternative names exported by the IO module: ioError and catch.

```
instance Monad IO where
    return a = ...   -- function from a -> IO a
    m >>= k  = ...   -- executes the I/O action m and binds the value to k's input
    fail s   = ioError (userError s)

data IOError = ...

ioError :: IOError -> IO a
ioError = ...

userError :: String -> IOError
userError = ...

catch :: IO a -> (IOError -> IO a) -> IO a
catch = ...

try :: IO a -> IO (Either IOError a)
try f = catch (do r <- f
                  return (Right r))
              (return . Left)
```

The IO monad is incorporated into the Monad Template Library framework as an instance of the MonadError class.

```
instance Error IOError where
  ...

instance MonadError IO where
    throwError = ioError
    catchError = catch
```

The IO module exports a convenience function called try that executes an I/O action and returns Right result if the action succeeded or Left IOError if an I/O error was caught.

Example

This example shows a partial implementation of the "tr" command that copies the standard input stream to the standard output stream with character translations controlled by command-line arguments. It demonstrates the use of the exception handling mechanisms of the MonadError class with the IO monad.

Code available in example14.hs

```
import Monad
import System
import IO
import Control.Monad.Error

-- translate char in set1 to corresponding char in set2
translate :: String -> String -> Char -> Char
translate []     _      c = c
translate (x:xs) []     c = if x == c then ' ' else translate xs []  c
translate (x:xs) [y]    c = if x == c then  y  else translate xs [y] c
translate (x:xs) (y:ys) c = if x == c then  y  else translate xs ys  c

-- translate an entire string
translateString :: String -> String -> String -> String
translateString set1 set2 str = map (translate set1 set2) str

usage :: IOError -> IO ()
usage e = do putStrLn "Usage: ex14 set1 set2"
             putStrLn "Translates characters in set1 on stdin to the corresponding"
             putStrLn "characters from set2 and writes the translation to stdout."

-- translates stdin to stdout based on commandline arguments
main :: IO ()
main = (do [set1,set2] <- getArgs
           contents    <- hGetContents stdin
           putStr $ translateString set1 set2 contents)
       `catchError` usage
```

Prev: The List monad TOC: Contents Next: The State monad

The State monad

- Overview
- Motivation
- Definition
- Example

Overview

Computation type:	Computations which maintain state.
Binding strategy:	Binding threads a state parameter through the sequence of bound functions so that the same state value is never used twice, giving the illusion of in-place update.
Useful for:	Building computations from sequences of operations that require a shared state.
Zero and plus:	None.
Example type:	State st a

Motivation

A pure functional language cannot update values in place because it violates referential transparency. A common idiom to simulate such stateful computations is to "thread" a state parameter through a sequence of functions:

```
data MyType = MT Int Bool Char Int deriving Show

makeRandomValue :: StdGen -> (MyType, StdGen)
makeRandomValue g = let (n,g1) = randomR (1,100) g
                        (b,g2) = random g1
                        (c,g3) = randomR ('a','z') g2
                        (m,g4) = randomR (-n,n) g3
                    in (MT n b c m, g4)
```

This approach works, but such code can be error-prone, messy and difficult to maintain. The State monad hides the threading of the state parameter inside the binding operation, simultaneously making the code easier to write, easier to read and easier to modify.

Definition

The definition shown here uses multi-parameter type classes and funDeps, which are not standard Haskell 98. It is not necessary to fully understand these details to make use of the State monad.

```
newtype State s a = State { runState :: (s -> (a,s)) }

instance Monad (State s) where
    return a        = State $ \s -> (a,s)
    (State x) >>= f = State $ \s -> let (v,s') = x s in runState (f v) s'
```

Values in the State monad are represented as transition functions from an initial state to a (value,newState) pair and a new type definition is provided to describe this construct: `State s a` is the type of a value of type `a` inside the State monad with state of type `s`.

The type constructor `State s` is an instance of the `Monad` class. The `return` function simply creates a state transition function which sets the value but leaves the state unchanged. The binding operator creates a state transition function that applies its right-hand argument to the value and new state from its left-hand argument.

```
class MonadState m s | m -> s where
    get :: m s
    put :: s -> m ()

instance MonadState (State s) s where
    get   = State $ \s -> (s,s)
    put s = State $ \_ -> ((),s)
```

The `MonadState` class provides a standard but very simple interface for State monads. The `get` function retrieves the state by copying it as the value. The `put` function sets the state of the monad and does not yield a value.

There are many additional functions provide which perform more complex computations built on top of `get` and put. The most useful one is `gets` which retrieves a function of the state. The others are listed in the documentation for the State monad library.

Example

A simple application of the State monad is to thread the random generator state through multiple calls to the generation function.

Code available in example15.hs

```
data MyType = MT Int Bool Char Int deriving Show

{- Using the State monad, we can define a function that returns
   a random value and updates the random generator state at
   the same time.
-}
getAny :: (Random a) => State StdGen a
getAny = do g      <- get
            (x,g') <- return $ random g
            put g'
            return x

-- similar to getAny, but it bounds the random value returned
getOne :: (Random a) => (a,a) -> State StdGen a
getOne bounds = do g      <- get
                   (x,g') <- return $ randomR bounds g
                   put g'
                   return x

{- Using the State monad with StdGen as the state, we can build
   random complex types without manually threading the
   random generator states through the code.
-}
makeRandomValueST :: StdGen -> (MyType, StdGen)
makeRandomValueST = runState (do n <- getOne (1,100)
                                 b <- getAny
                                 c <- getOne ('a','z')
                                 m <- getOne (-n,n)
                                 return (MT n b c m))
```

Prev: The IO monad TOC: Contents Next: The Reader monad

The Reader monad

Overview

Computation type:	Computations which read values from a shared environment.
Binding strategy:	Monad values are functions from the environment to a value. The bound function is applied to the bound value, and both have access to the shared environment.
Useful for:	Maintaining variable bindings, or other shared environment.
Zero and plus:	None.
Example type:	Reader [(String,Value)] a

Motivation

Some programming problems require computations within a shared environment (such as a set of variable bindings). These computations typically read values from the environment and sometimes execute sub-computations in a modified environment (with new or shadowing bindings, for example), but they do not require the full generality of the State monad.

The Reader monad is specifically designed for these types of computations and is often a clearer and easier mechanism than using the State monad.

Definition

The definition shown here uses multi-parameter type classes and funDeps, which are not standard Haskell 98. It is not necessary to fully understand these details to make use of the Reader monad.

```
newtype Reader e a = Reader { runReader :: (e -> a) }

instance Monad (Reader e) where
    return a         = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> f (r e) e
```

Values in the Reader monad are functions from an environment to a value. To extract the final value from a computation in the Reader monad, you simply apply `(runReader reader)` to an environment value.

The `return` function creates a `Reader` that ignores the environment and produces the given value. The binding operator produces a `Reader` that uses the environment to extract the value its left-hand side and then applies the bound function to that value in the same environment.

```
class MonadReader e m | m -> e where
    ask   :: m e
    local :: (e -> e) -> m a -> m a

instance MonadReader (Reader e) where
    ask       = Reader id
    local f c = Reader $ \e -> runReader c (f e)

asks :: (MonadReader e m) => (e -> a) -> m a
asks sel = ask >>= return . sel
```

The `MonadReader` class provides a number of convenience functions that are very useful when working with a Reader monad. The `ask` function retrieves the environment and the `local` function executes a computation in a modified environment. The `asks` function is a convenience function that retrieves a function of the current environment, and is typically used with a selector or lookup function.

Example

Consider the problem of instantiating templates which contain variable substitutions and included templates. Using the Reader monad, we can maintain an environment of all known templates and all known variable bindings. Then, when a variable substitution is encountered, we can use the `asks` function to lookup the value of the variable. When a template is included with new variable definitions, we can use the `local` function to resolve the template in a modified environment that contains the additional variable bindings.

Code available in example16.hs

```
-- This the abstract syntax representation of a template
--              Text       Variable     Quote        Include                  Compound
data Template = T String | V Template | Q Template | I Template [Definition] | C [Template]
data Definition = D Template Template

-- Our environment consists of an association list of named templates and
-- an association list of named variable values.
data Environment = Env {templates::[(String,Template)],
                        variables::[(String,String)]}

-- lookup a variable from the environment
lookupVar :: String -> Environment -> Maybe String
lookupVar name env = lookup name (variables env)

-- lookup a template from the environment
lookupTemplate :: String -> Environment -> Maybe Template
lookupTemplate name env = lookup name (templates env)

-- add a list of resolved definitions to the environment
addDefs :: [(String,String)] -> Environment -> Environment
addDefs defs env = env {variables = defs ++ (variables env)}

-- resolve a Definition and produce a (name,value) pair
resolveDef :: Definition -> Reader Environment (String,String)
resolveDef (D t d) = do name <- resolve t
                        value <- resolve d
                        return (name,value)

-- resolve a template into a string
resolve :: Template -> Reader Environment (String)
resolve (T s)    = return s
resolve (V t)    = do varName  <- resolve t
                      varValue <- asks (lookupVar varName)
        return $ maybe "" id varValue
resolve (Q t)    = do tmplName <- resolve t
                      body     <- asks (lookupTemplate tmplName)
                      return $ maybe "" show body
resolve (I t ds) = do tmplName <- resolve t
                      body     <- asks (lookupTemplate tmplName)
                      case body of
                        Just t' -> do defs <- mapM resolveDef ds
                                      local (addDefs defs) (resolve t')
                        Nothing -> return ""
resolve (C ts)   = (liftM concat) (mapM resolve ts)
```

To use the Reader monad to resolve a template `t` into a `String`, you simply need to do `runReader (resolve t) env`.

Prev: The State monad TOC: Contents Next: The Writer monad

The Writer monad

Overview

Computation type:	Computations which produce a stream of data in addition to the computed values.
Binding strategy:	A Writer monad value is a (computation value, log value) pair. Binding replaces the computation value with the result of applying the bound function to the previous value and appends any log data from the computation to the existing log data.
Useful for:	Logging, or other computations that produce output "on the side".
Zero and plus:	None.
Example type:	Writer [String] a

Motivation

It is often desirable for a computation to generate output "on the side". Logging and tracing are the most common examples in which data is generated during a computation that we want to retain but is not the primary result of the computation.

Explicitly managing the logging or tracing data can clutter up the code and invite subtle bugs such as missed log entries. The Writer monad provides a cleaner way to manage the output without cluttering the main computation.

Definition

The definition shown here uses multi-parameter type classes and funDeps, which are not standard Haskell 98. It is not necessary to fully understand these details to make use of the Writer monad.

To fully understand this definition, you need to know about Haskell's `Monoid` class, which represents a mathematical structure called a monoid in the same way that the `Monad` class represents the monad structure.

The good news is that monoids are simpler than monads. A monoid is a set of objects, a single identity element, and an associative binary operator over the set of objects. A monoid must obey some mathematical laws, such that applying the operator to any values from the set gives another value in the set, and whenever one operand of the operator is the identity element the result is equal to the other operand. You may notice that these laws are the same as the laws governing `mzero` and `mplus` for instances of `MonadPlus`. That is because monads with a zero and plus are monads that are also monoids!

Some examples of mathematical monoids are the natural numbers with identity element 0 and binary operator for addition, and also the natural numbers with identity element 1 and binary operator for multiplication.

In Haskell, a monoid consists of a type, an identity element, and a binary operator. Haskell defines the `Monoid` class (in Data.Monoid) to provide a standard convention for working with monoids: the identity element is named `mempty` and the operator is named `mappend`.

The most commonly used standard monoid in Haskell is the list, but functions of type `(a -> a)` also form a monoid.

Care should be taken when using a list as the monoid for a Writer, as there may be a performance penalty associated with the `mappend` operation as the output grows. In that case, a data structure that supports fast append operations would be a more appropriate choice.

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

instance (Monoid w) => Monad (Writer w) where
    return a             = Writer (a,mempty)
    (Writer (a,w)) >>= f = let (a’,w’) = runWriter $ f a in Writer (a’,w ‘mappend‘ w’)
```

The Writer monad maintains a (value,log) pair, where the log type must be a monoid. The `return` function simply returns the value along with an empty log. Binding executes the bound function using the current value as input, and appends any log output to the existing log.

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    pass   :: m (a,w -> w) -> m a
    listen :: m a -> m (a,w)
    tell   :: w -> m ()

instance (Monoid w) => MonadWriter (Writer w) where
    pass   (Writer ((a,f),w)) = Writer (a,f w)
    listen (Writer (a,w))     = Writer ((a,w),w)
    tell   s                  = Writer ((),s)

listens :: (MonadWriter w m) => (w -> w) -> m a -> m (a,w)
listens f m = do (a,w) <- m; return (a,f w)

censor :: (MonadWriter w m) => (w -> w) -> m a -> m a
censor f m = pass $ do a <- m; return (a,f)
```

The `MonadWriter` class provides a number of convenience functions for working with Writer monads. The simplest and most useful is `tell`, which adds one or more entries to the log. The `listen` function turns a Writer that returns a value `a` and produces output `w` into a Writer that produces a value `(a,w)` and still produces output `w`. This allows the computation to "listen" to the log output generated by a Writer.

The `pass` function is slightly more complicated. It converts a Writer that produces a value `(a,f)` and output `w` into a Writer that produces a value `a` and output `f w`. This is somewhat cumbersome, so the helper function `censor` is normally used. The `censor` function takes a function and a Writer and produces a new Writer whose output is the same but whose log entry has been modified by the function.

The `listens` function operates just like `listen` except that the log part of the value is modified by the supplied function.

Example

In this example, we imagine a very simple firewall that filters packets based on a rulebase of rules matching the source and destination hosts and the payload of the packet. The firewall's primary job is packet filtering, but we would also like it to produce a log of its activity.

Code available in example17.hs

```
-- this is the format of our log entries
data Entry = Log {count::Int, msg::String} deriving Eq

-- add a message to the log
logMsg :: String -> Writer [Entry] ()
logMsg s = tell [Log 1 s]

-- this handles one packet
filterOne :: [Rule] -> Packet -> Writer [Entry] (Maybe Packet)
filterOne rules packet = do rule <- return (match rules packet)
                            case rule of
                              Nothing  -> do logMsg ("DROPPING UNMATCHED PACKET: " ++ (show packet))
                                             return Nothing
                              (Just r) -> do when (logIt r) (logMsg ("MATCH: " ++ (show r) ++ " <=> " ++ (show packet)))
                                             case r of
                                               (Rule Accept _ _) -> return (Just packet)
                                               (Rule Reject _ _) -> return Nothing
```

That was pretty simple, but what if we want to merge duplicate consecutive log entries? None of the existing functions allow us to modify the output from previous stages of the computation, but we can use a "delayed logging" trick to only add a log entry only after we get a new entry that doesn't match the ones before it.

Code available in example17.hs

```
-- merge identical entries at the end of the log
-- This function uses [Entry] as both the log type and the result type.
-- When two identical messages are merged, the result is just the message
-- with an incremented count.  When two different messages are merged,
-- the first message is logged and the second is returned as the result.
mergeEntries :: [Entry] -> [Entry] -> Writer [Entry] [Entry]
mergeEntries []   x    = return x
mergeEntries x    []   = return x
mergeEntries [e1] [e2] = let (Log n msg)   = e1
                             (Log n' msg') = e2
                         in if msg == msg' then
                              return [(Log (n+n') msg)]
                            else
                              do tell [e1]
                                 return [e2]

-- This is a complex-looking function but it is actually pretty simple.
-- It maps a function over a list of values to get a list of Writers,
-- then runs each writer and combines the results.  The result of the function
-- is a writer whose value is a list of all the values from the writers and whose
-- log output is the result of folding the merge operator into the individual
-- log entries (using 'initial' as the initial log value).
groupSame :: (Monoid a) => a -> (a -> a -> Writer a a) -> [b] -> (b -> Writer a c) -> Writer a [c]
groupSame initial merge []     _  = do tell initial
                                       return []
groupSame initial merge (x:xs) fn = do (result,output) <- return (runWriter (fn x))
                                       new             <- merge initial output
                                       rest            <- groupSame new merge xs fn
                                       return (result:rest)

-- this filters a list of packets, producing a filtered packet list and a log of
-- the activity in which consecutive messages are merged
filterAll :: [Rule] -> [Packet] -> Writer [Entry] [Packet]
filterAll rules packets = do tell [Log 1 "STARTING PACKET FILTER"]
                             out <- groupSame [] mergeEntries packets (filterOne rules)
                             tell [Log 1 "STOPPING PACKET FILTER"]
                             return (catMaybes out)
```

The Continuation monad

Overview

Computation type:	Computations which can be interrupted and resumed.
Binding strategy:	Binding a function to a monadic value creates a new continuation which uses the function as the continuation of the monadic computation.
Useful for:	Complex control structures, error handling and creating co-routines.
Zero and plus:	None.
Example type:	Cont r a

Motivation

Abuse of the Continuation monad can produce code that is impossible to understand and maintain.

Before using the Continuation monad, be sure that you have a firm understanding of continuation-passing style (CPS) and that continuations represent the best solution to your particular design problem. Many algorithms which require continuations in other languages do not require them in Haskell, due to Haskell's lazy semantics.

Continuations represent the *future* of a computation, as a function from an intermediate result to the final result. In continuation-passing style, computations are built up from sequences of nested continuations, terminated by a final continuation (often `id`) which produces the final result. Since continuations are functions which represent the future of a computation, manipulation of the continuation functions can achieve complex manipulations of the future of the computation, such as interrupting a computation in the middle, aborting a portion of a computation, restarting a computation and interleaving execution of computations. The Continuation monad adapts CPS to the structure of a monad.

Definition

```
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) } -- r is the final result type of the whole computation

instance Monad (Cont r) where
    return a       = Cont $ \k -> k a                       -- i.e. return a = \k -> k a
    (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k) -- i.e. c >>= f = \k -> c (\a -> f a k)
```

The Continuation monad represents computations in continuation-passing style. `Cont r a` is a CPS computation that produces an intermediate result of type `a` within a CPS computation whose final result type is `r`.

The `return` function simply creates a continuation which passes the value on. The >>= operator adds the bound function into the continuation chain.

```
class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a

instance MonadCont (Cont r) where
    callCC f = Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

The `MonadCont` class provides the `callCC` function, which provides an *escape* continuation mechanism for use with Continuation monads. Escape continuations allow you to abort the current computation and return a value immediately. They achieve a similar effect to `throwError` and catchError within an `Error` monad.

`callCC` calls a function with the current continuation as its argument (hence the name). The standard idiom used with `callCC` is to provide a lambda-expression to name the continuation. Then calling the named continuation anywhere within its scope will escape from the computation, even if it is many layers deep within nested computations.

In addition to the escape mechanism provided by `callCC`, the Continuation monad can be used to implement other, more powerful continuation manipulations. These other mechanisms have fairly specialized uses, however — and abuse of them can easily create fiendishly obfuscated code — so they will not be covered here.

Example

This example gives a taste of how escape continuations work. The example function uses escape continuations to perform a complicated transformation on an integer.

Code available in example18.hs

```
{- We use the continuation monad to perform "escapes" from code blocks.
   This function implements a complicated control structure to process
   numbers:

   Input (n)       Output                       List Shown
   =========       ======                       ==========
   0-9             n                            none
   10-199          number of digits in (n/2)    digits of (n/2)
   200-19999       n                            digits of (n/2)
   20000-1999999   (n/2) backwards              none
   >= 2000000      sum of digits of (n/2)       digits of (n/2)
-}
fun :: Int -> String
fun n = (`runCont` id) $ do
        str <- callCC $ \exit1 -> do                        -- define "exit1"
          when (n < 10) (exit1 (show n))
          let ns = map digitToInt (show (n `div` 2))
          n' <- callCC $ \exit2 -> do                       -- define "exit2"
            when ((length ns) < 3) (exit2 (length ns))
            when ((length ns) < 5) (exit2 n)
            when ((length ns) < 7) $ do let ns' = map intToDigit (reverse ns)
                                        exit1 (dropWhile (=='0') ns')  --escape 2 levels
            return $ sum ns
          return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
        return $ "Answer: " ++ str
```

Prev: The Writer monad TOC: Contents Next: Part III - Introduction

Introduction

Part I has introduced the monad concept and Part II has provided an understanding of a number of common, useful monads in the standard Haskell libraries. This is not enough to put monads into heavy practice, however, because in the real world you often want computations which combine aspects of more than one monad at the same time, such as stateful, non-determistic computations or computations which make use of continuations and perform I/O. When one computation is a strict subset of the other, it is possible to perform the monad computations separately, unless the sub-computation is performed in a one-way monad.

Often, the computations can't be performed in isolation. In this case, what is needed is a monad that combines the features of the two monads into a single computation. It is inefficient and poor practice to write a new monad instance with the required characteristics each time a new combination is desired. Instead, we would prefer to develop a way to combine the standard monads to produce the needed hybrids. The technique that lets us do exactly that is called *monad transformers*.

Monad transformers are the topic of Part III, and they are explained by revisiting earlier examples to see how monad transformers can be used to add more realistic capabilities to them. It may be helpful to review the earlier examples as they are re-examined.

Combining monads the hard way

- Nested Monads
- Combined Monads

Before we investigate the use of monad transformers, we will see how monads can be combined without using transformers. This is a useful excercise to develop insights into the issues that arise when combining monads and provides a baseline from which the advantages of the transformer approach can be measured. We use the code from example 18 (the Continuation monad) to illustrate these issues, so you may want to review it before continuing.

Nested Monads

Some computations have a simple enough structure that the monadic computations can be nested, avoiding the need for a combined monad altogether. In Haskell, all computations occur in the IO monad at the top level, so the monad examples we have seen so far all actually use the technique of nested monadic computations. To do this, the computations perform all of their input at the beginning — usually by reading arguments from the command line — then pass the values on to the monadic computations to produce results, and finally perform their output at the end. This structure avoids the issues of combining monads but makes the examples seem contrived at times.

The code introduced in example 18 followed the nesting pattern: reading a number from the command line in the IO monad, passing that number to a computation in the Continuation monad to produce a string, and then writing the string back in the IO monad. The computations in the IO monad aren't restricted to reading from the command line and writing strings; they can be arbitrarily complex. Likewise, the inner computation can be arbitrarily complex as well. As long as the inner computation does not depend on the functionality of the outer monad, it can be safely nested within the outer monad, as illustrated in this variation on example 18 which reads the value from stdin instead of using a command line argument:

Code available in example19.hs

```
fun :: IO String
fun = do n <- (readLn::IO Int)         -- this is an IO monad block
         return $ (`runCont` id) $ do  -- this is a Cont monad block
           str <- callCC $ \exit1 -> do
             when (n < 10) (exit1 (show n))
             let ns = map digitToInt (show (n `div` 2))
             n' <- callCC $ \exit2 -> do
               when ((length ns) < 3) (exit2 (length ns))
               when ((length ns) < 5) (exit2 n)
               when ((length ns) < 7) $ do let ns' = map intToDigit (reverse ns)
                                           exit1 (dropWhile (=='0') ns')
               return $ sum ns
             return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
           return $ "Answer: " ++ str
```

Combined Monads

What about computations with more complicated structure? If the nesting pattern cannot be used, we need a way to combine the attributes of two or more monads in a single computation. This is accomplished by doing computations within a monad in which the values are themselves monadic values in another monad. For example, we might perform computations in the Continuation monad of type `Cont (IO String) a` if we need to perform I/O within the computation in the Continuation monad. We could use a monad of type `State (Either Err a) a` to combine the features of the State and Error monads in a single computation.

Consider a slight modification to our example in which we perform the same I/O at the beginning, but we may require additional input in the middle of the computation in the Continuation monad. In this case, we will allow the user to specify part of the output value when the input value is within a certain range. Because the I/O depends on part of the computation in the Continuation monad and part of the computation in the Continuation monad depends on the result of the I/O, we cannot use the nested monad pattern.

Instead, we make the computation in the Continuation monad use values from the IO monad. What used to be `Int` and `String` values are now of type `IO Int` and `IO String`. We can't extract values from the IO monad — it's a one-way monad — so we may need to nest little do-blocks of the IO monad within the Continuation monad to manipulate the values. We use a helper function `toIO` to make it clearer when we are creating values in the IO monad nested within the Continuation monad.

Code available in example20.hs

```
toIO :: a -> IO a
toIO x = return x

fun :: IO String
fun = do n <- (readLn::IO Int)         -- this is an IO monad block
         convert n

convert :: Int -> IO String
convert n = (`runCont` id) $ do        -- this is a Cont monad block
              str <- callCC $ \exit1 -> do    -- str has type IO String
                when (n < 10) (exit1 $ toIO (show n))
                let ns = map digitToInt (show (n `div` 2))
                n' <- callCC $ \exit2 -> do   -- n' has type IO Int
                  when ((length ns) < 3) (exit2 (toIO (length ns)))
                  when ((length ns) < 5) (exit2 $ do putStrLn "Enter a number:"
                                                     x <- (readLn::IO Int)
                                                     return x)
                  when ((length ns) < 7) $ do let ns' = map intToDigit (reverse ns)
                                              exit1 $ toIO (dropWhile (=='0') ns')
                  return (toIO (sum ns))
                return $ do num <- n'  -- this is an IO monad block
                            return $ "(ns = " ++ (show ns) ++ ") " ++ (show num)
              return $ do s <- str -- this is an IO monad block
                          return $ "Answer: " ++ s
```

Even this trivial example has gotten confusing and ugly when we tried to combine different monads in the same computation. It works, but it isn't pretty. Comparing the code side-by-side shows the degree to which the manual monad combination strategy pollutes the code.

<table>
<tr><th>Nested monads from example 19</th><th>Manually combined monads from example 20</th></tr>
<tr><td>

```
fun = do n <- (readLn::IO Int)
         return $ (`runCont` id) $ do
           str <- callCC $ \exit1 -> do
             when (n < 10) (exit1 (show n))
             let ns = map digitToInt (show (n `div` 2))
             n' <- callCC $ \exit2 -> do
               when ((length ns) < 3) (exit2 (length ns))
               when ((length ns) < 5) (exit2 n)
               when ((length ns) < 7) $ do
                 let ns' = map intToDigit (reverse ns)
                 exit1 (dropWhile (=='0') ns')
               return $ sum ns
             return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
           return $ "Answer: " ++ str
```

</td><td>

```
convert n = (`runCont` id) $ do
              str <- callCC $ \exit1 -> do
                when (n < 10) (exit1 $ toIO (show n))
                let ns = map digitToInt (show (n `div` 2))
                n' <- callCC $ \exit2 -> do
                  when ((length ns) < 3) (exit2 (toIO (length ns)))
                  when ((length ns) < 5) (exit2 $ do
                        putStrLn "Enter a number:"
                        x <- (readLn::IO Int)
                        return x)
                  when ((length ns) < 7) $ do
                    let ns' = map intToDigit (reverse ns)
                    exit1 $ toIO (dropWhile (=='0') ns')
                  return (toIO (sum ns))
                return $ do num <- n'
                            return $ "(ns = " ++ (show ns) ++ ") " ++ (show num)
              return $ do s <- str
                          return $ "Answer: " ++ s
```

</td></tr>
</table>

Prev: Part III - Introduction TOC: Contents Next: Monad transformers

Monad transformers

Monad transformers are special variants of standard monads that facilitate the combining of monads. Their type constructors are parameterized over a monad type constructor, and they produce combined monadic types.

Transformer type constructors

Type constructors play a fundamental role in Haskell's monad support. Recall that `Reader r a` is the type of values of type `a` within a Reader monad with environment of type `r`. The type constructor `Reader r` is an instance of the `Monad` class, and the `runReader::(r->a)` function performs a computation in the Reader monad and returns the result of type `a`.

A transformer version of the Reader monad, called `ReaderT`, exists which adds a monad type constructor as an addition parameter. `ReaderT r m a` is the type of values of the combined monad in which Reader is the *base monad* and `m` is the *inner monad*. `ReaderT r m` is an instance of the monad class, and the `runReaderT::(r -> m a)` function performs a computation in the combined monad and returns a result of type `m a`.

Using the transformer versions of the monads, we can produce combined monads very simply. `ReaderT r IO` is a combined Reader+IO monad. We can also generate the non-transformer version of a monad from the transformer version by applying it to the Identity monad. So `ReaderT r Identity` is the same monad as `Reader r`.

If your code produces kind errors during compilation, it means that you are not using the type cosntructors properly. Make sure that you have supplied the correct number of parameters to the type constructors and that you have not left out any parenthesis in complex type expressions.

Lifting

When using combined monads created by the monad transformers, we avoid having to explicitly manage the inner monad types, resulting in clearer, simpler code. Instead of creating additional do-blocks within the computation to manipulate values in the inner monad type, we can use lifting operations to bring functions from the inner monad into the combined monad.

Recall the `liftM` family of functions which are used to lift non-monadic functions into a monad. Each monad transformer provides a `lift` function that is used to lift a monadic computation into a combined monad. Many transformers also provide a `liftIO` function, which is a version of `lift` that is optimized for lifting computations in the `IO` monad. To see this in action, we will continue to develop our previous example in the Continuation monad.

Code available in example21.hs

```
fun :: IO String
fun = (`runContT` return) $ do
        n   <- liftIO (readLn::IO Int)
        str <- callCC $ \exit1 -> do     -- define "exit1"
          when (n < 10) (exit1 (show n))
          let ns = map digitToInt (show (n `div` 2))
          n' <- callCC $ \exit2 -> do    -- define "exit2"
            when ((length ns) < 3) (exit2 (length ns))
            when ((length ns) < 5) $ do liftIO $ putStrLn "Enter a number:"
                                        x <- liftIO (readLn::IO Int)
                                        exit2 x
            when ((length ns) < 7) $ do let ns' = map intToDigit (reverse ns)
                                        exit1 (dropWhile (=='0') ns')  --escape 2 levels
            return $ sum ns
          return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
        return $ "Answer: " ++ str
```

Compare this function using `ContT`, the transformer version of `Cont`, with the original version to see how unobtrusive the changes become when using the monad transformer.

Nested monads from example 19	**Monads combined with a transformer from example 21**

`fun = do n <- (readLn::IO Int)`
`return $ (`runCont` **id**) $ do`
`str <- callCC $ \exit1 -> do`
`when (n < 10) (exit1 (show n))`
`let ns = map digitToInt (show (n `div` 2))`
`n' <- callCC $ \exit2 -> do`
`when ((length ns) < 3) (exit2 (length ns))`
`when ((length ns) < 5) (exit2 **n**)`
`when ((length ns) < 7) $ do`
`let ns' = map intToDigit (reverse ns)`
`exit1 (dropWhile (=='0') ns')`
`return $ sum ns`
`return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')`
`return $ "Answer: " ++ str`
	`fun = (`runCont**T**` **return**) $ do`
	`n <- **liftIO** (readLn::IO Int)`
	`str <- callCC $ \exit1 -> do`
	`when (n < 10) (exit1 (show n))`
	`let ns = map digitToInt (show (n `div` 2))`
	`n' <- callCC $ \exit2 -> do`
	`when ((length ns) < 3) (exit2 (length ns))`
	`when ((length ns) < 5) $ **do**`
	`**liftIO $ putStrLn "Enter a number:"**`
	`**x <- liftIO (readLn::IO Int)**`
	`exit2 **x**`
	`when ((length ns) < 7) $ do`
	`let ns' = map intToDigit (reverse ns)`
	`exit1 (dropWhile (=='0') ns')`
	`return $ sum ns`
	`return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')`
	`return $ "Answer: " ++ str`

The impact of adding the I/O in the middle of the computation is narrowly confined when using the monad transformer. Contrast this with the changes required to achieve the same result using a manually combined monad.

Prev: Combining monads the hard way

TOC: Contents

Next: Standard monad transformers

Standard monad transformers

- The MonadTrans and MonadIO classes
- Transformer versions of standard monads

Haskell's base libraries provide support for monad transformers in the form of classes which represent monad transformers and special transformer versions of standard monads.

The MonadTrans and MonadIO classes

The `MonadTrans` class is defined in Control.Monad.Trans and provides the single function `lift`. The `lift` function lifts a monadic computation in the inner monad into the combined monad.

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

Monads which provide optimized support for lifting IO operations are defined as members of the `MonadIO` class, which defines the `liftIO` function.

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

Transformer versions of standard monads

The standard monads of the monad template library all have transformer versions which are defined consistently with their non-transformer versions. However, it is not the case the all monad transformers apply the same transformation. We have seen that the `ContT` transformer turns continuations of the form `(a->r)->r` into continuations of the form `(a->m r)->m r`. The `StateT` transformer is different. It turns state transformer functions of the form `s->(a,s)` into state transformer functions of the form `s->m (a,s)`. In general, there is no magic formula to create a transformer version of a monad — the form of each transformer depends on what makes sense in the context of its non-transformer type.

Standard Monad	Transformer Version	Original Type	Combined Type
Error	ErrorT	`Either e a`	`m (Either e a)`
State	StateT	`s -> (a,s)`	`s -> m (a,s)`
Reader	ReaderT	`r -> a`	`r -> m a`
Writer	WriterT	`(a,w)`	`m (a,w)`
Cont	ContT	`(a -> r) -> r`	`(a -> m r) -> m r`

Order is important when combining monads. `StateT s (Error e)` is different than `ErrorT e (State s)`. The first produces a combined type of `s -> Error e (a,s)`, in which the computation can either return a new state or generate an error. The second combination produces a combined type of `s -> (Error e a,s)`, in which the computation always returns a new state, and the value can be an error or a normal value.

Prev: Standard monad transformers	TOC: Contents	Next: Anatomy of a monad transformer

Anatomy of a monad transformer

- Combined monad definition
- Defining the lifting function
- Functors

In this section, we will take a detailed look at the implementation of one of the more interesting transformers in the standard library, `StateT`. Studying this transformer will build insight into the transformer mechanism that you can call upon when using monad transformers in your code. You might want to review the section on the State monad before continuing.

Combined monad definition

Just as the State monad was built upon the definition

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

the StateT transformer is built upon the definition

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

`State s` is an instance of both the `Monad` class and the `MonadState s` class, so `StateT s m` should also be members of the `Monad` and `MonadState s` classes. Furthermore, if `m` is an instance of `MonadPlus`, `StateT s m` should also be a member of `MonadPlus`.

To define `StateT s m` as a `Monad` instance:

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }

instance (Monad m) => Monad (StateT s m) where
    return a          = StateT $ \s -> return (a,s)
    (StateT x) >>= f  = StateT $ \s -> do (v,s')      <- x s            -- get new value, state
                                          (StateT x') <- return $ f v   -- apply bound function to get new state transformation fn
                                          x' s'                         -- apply the state transformation fn to the new state
```

Compare this to the definition for `State s`. Our definition of `return` makes use of the `return` function of the inner monad, and the binding operator uses a do-block to perform a computation in the inner monad.

We also want to declare all combined monads that use the `StateT` transformer to be instaces of the `MonadState` class, so we will have to give definitions for `get` and `put`:

```
instance (Monad m) => MonadState s (StateT s m) where
    get   = StateT $ \s -> return (s,s)
    put s = StateT $ \_ -> return ((),s)
```

Finally, we want to declare all combined monads in which `StateT` is used with an instance of `MonadPlus` to be instances of `MonadPlus`:

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
    mzero = StateT $ \s -> mzero
    (StateT x1) `mplus` (StateT x2) = StateT $ \s -> (x1 s) `mplus` (x2 s)
```

Defining the lifting function

The final step to make our monad transformer fully integrated with Haskell's monad classes is to make `StateT s` an instance of the `MonadTrans` class by providing a `lift` function:

```
instance MonadTrans (StateT s) where
    lift c = StateT $ \s -> c >>= (\x -> return (x,s))
```

The `lift` function creates a `StateT` state transformation function that binds the computation in the inner monad to a function that packages the result with the input state. The result is that a function that returns a list (i.e., a computation in the List monad) can be lifted into `StateT s []`, where it becomes a function that returns a `StateT (s -> [(a,s)])`. That is, the lifted computation produces *multiple* (value,state) pairs from its input state. The effect of this is to "fork" the computation in StateT, creating a different branch of the computation for each value in the list returned by the lifted function. Of course, applying `StateT` to a different monad will produce different semantics for the `lift` function.

Functors

We have examined the implementation of one monad transformer above, and it was stated earlier that there was no magic formula to produce transformer versions of monads. Each transformer's implementation will depend on the nature of the computational effects it is adding to the inner monad.

Despite this, there is some theoretical foundation to the theory of monad transformers. Certain transformers can be grouped according to how they use the inner monad, and the transformers within each group can be derived using monadic functions and *functors*. Functors, roughly, are types which support a mapping operation `fmap :: (a->b) -> f a -> f b`. To learn more about it, check out Mark Jones' influential paper that inspired the Haskell monad template library.

Prev: Standard monad transformers | TOC: Contents | Next: More examples with monad transformers

More examples with monad transformers

- WriterT with IO
- ReaderT with IO
- StateT with List

At this point, you should know everything you need to begin using monads and monad transformers in your programs. The best way to build proficiency is to work on actual code. As your monadic programs become more abitious, you may find it awkward to mix additional transformers into your combined monads. This will be addressed in the next section, but first you should master the basic process of applying a single transformer to a base monad.

WriterT with IO

Try adapting the firewall simulator of example 17 to include a timestamp on each log entry (don't worry about merging entries). The necessary changes should look something like this:

Code available in example22.hs

```
-- this is the format of our log entries
data Entry = Log {timestamp::ClockTime, msg::String} deriving Eq

instance Show Entry where
  show (Log t s) = (show t) ++ " | " ++ s

-- this is the combined monad type
type LogWriter a = WriterT [Entry] IO a

-- add a message to the log
logMsg :: String -> LogWriter ()
logMsg s = do t <- liftIO getClockTime
              tell [Log t s]

-- this handles one packet
filterOne :: [Rule] -> Packet -> LogWriter (Maybe Packet)
filterOne rules packet = do rule <- return (match rules packet)
                            case rule of
                              Nothing  -> do logMsg ("DROPPING UNMATCHED PACKET: " ++ (show packet))
                                             return Nothing
                              (Just r) -> do when (logIt r) (logMsg ("MATCH: " ++ (show r) ++ " <=> " ++ (show packet)))
                                             case r of
                                               (Rule Accept _ _) -> return (Just packet)
                                               (Rule Reject _ _) -> return Nothing

-- this filters a list of packets, producing a filtered packet list
-- and a log of the activity
filterAll :: [Rule] -> [Packet] -> LogWriter [Packet]
filterAll rules packets = do logMsg "STARTING PACKET FILTER"
                             out <- mapM (filterOne rules) packets
                             logMsg "STOPPING PACKET FILTER"
                             return (catMaybes out)

-- read the rule data from the file named in the first argument, and the packet data from
-- the file named in the second argument, and then print the accepted packets followed by
-- a log generated during the computation.
main :: IO ()
main = do args       <- getArgs
          ruleData   <- readFile (args!!0)
          packetData <- readFile (args!!1)
          let rules   = (read ruleData)::[Rule]
              packets = (read packetData)::[Packet]
          (out,log) <- runWriterT (filterAll rules packets)
          putStrLn "ACCEPTED PACKETS"
          putStr (unlines (map show out))
          putStrLn "\n\nFIREWALL LOG"
          putStr (unlines (map show log))
```

ReaderT with IO

If you found that one too easy, move on to a slightly more complex example: convert the template system in example 16 from using a single template file with named templates to treating individual files as templates. One possible solution is shown in example 23, but try to do it without looking first.

StateT with List

The previous examples have all been using the IO monad as the inner monad. Here is a more interesting example: combining `StateT` with the List monad to produce a monad for stateful nondeterministic computations.

We will apply this powerful monad combination to the task of solving constraint satisfaction problems (in this case, a logic problem). The idea behind it is to have a number of variables that can take on different values and a number of predicates involving those variables that must be satisfied. The current variable assignments and the predicates make up the state of the computation, and the non-deterministic nature of the List monad allows us to easily test all combinations of variable assignments.

We start by laying the groundwork we will need to represent the logic problem, a simple predicate language:

Code available in example24.hs

```
-- First, we develop a language to express logic problems
type Var   = String
type Value = String
data Predicate = Is    Var Value            -- var has specific value
               | Equal Var Var              -- vars have same (unspecified) value
        | And   Predicate Predicate  -- both are true
        | Or    Predicate Predicate  -- at least one is true
        | Not   Predicate            -- it is not true
  deriving (Eq, Show)

type Variables = [(Var,Value)]

-- test for a variable NOT equaling a value
isNot :: Var -> Value -> Predicate
isNot var value = Not (Is var value)

-- if a is true, then b must also be true
implies :: Predicate -> Predicate -> Predicate
implies a b = Not (a `And` (Not b))

-- exclusive or
orElse :: Predicate -> Predicate -> Predicate
orElse a b = (a `And` (Not b)) `Or` ((Not a) `And` b)

-- Check a predicate with the given variable bindings.
-- An unbound variable causes a Nothing return value.
check :: Predicate -> Variables -> Maybe Bool
check (Is var value) vars = do val <- lookup var vars
                               return (val == value)
check (Equal v1 v2)  vars = do val1 <- lookup v1 vars
                               val2 <- lookup v2 vars
          return (val1 == val2)
check (And p1 p2)    vars = liftM2 (&&) (check p1 vars) (check p2 vars)
check (Or  p1 p2)    vars = liftM2 (||) (check p1 vars) (check p2 vars)
check (Not p)        vars = liftM (not) (check p vars)
```

The next thing we will need is some code for representing and solving constraint satisfaction problems. This is where we will define our combined monad.

Code available in example24.hs

```
-- this is the type of our logic problem
data ProblemState = PS {vars::Variables, constraints::[Predicate]}

-- this is our monad type for non-determinstic computations with state
type NDS a = StateT ProblemState [] a

-- lookup a variable
getVar :: Var -> NDS (Maybe Value)
getVar v = do vs <- gets vars
              return $ lookup v vs

-- set a variable
setVar :: Var -> Value -> NDS ()
setVar v x = do st <- get
                vs' <- return $ filter ((v/=).fst) (vars st)
                put $ st {vars=(v,x):vs'}

-- Check if the variable assignments satisfy all of the predicates.
-- The partial argument determines the value used when a predicate returns
-- Nothing because some variable it uses is not set.  Setting this to True
-- allows us to accept partial solutions, then we can use a value of
-- False at the end to signify that all solutions should be complete.
isConsistent :: Bool -> NDS Bool
isConsistent partial = do cs <- gets constraints
                          vs <- gets vars
                          let results = map (\p->check p vs) cs
                          return $ and (map (maybe partial id) results)

-- Return only the variable bindings that are complete consistent solutions.
getFinalVars :: NDS Variables
getFinalVars = do c <- isConsistent False
                  guard c
                  gets vars

-- Get the first solution to the problem, by evaluating the solver computation with
-- an initial problem state and then returning the first solution in the result list,
-- or Nothing if there was no solution.
getSolution :: NDS a -> ProblemState -> Maybe a
getSolution c i = listToMaybe (evalStateT c i)

-- Get a list of all possible solutions to the problem by evaluating the solver
-- computation with an initial problem state.
getAllSolutions :: NDS a -> ProblemState -> [a]
getAllSolutions c i = evalStateT c i
```

We are ready to apply the predicate language and stateful nondeterministic monad to solving a logic problem. For this example, we will use the well-known Kalotan puzzle which appeared in *Mathematical Brain-Teasers*, Dover Publications (1976), by J. A. H. Hunter.

> The Kalotans are a tribe with a peculiar quirk: their males always tell the truth. Their females never make two consecutive true statements, or two consecutive untrue statements. An anthropologist (let's call him Worf) has begun to study them. Worf does not yet know the Kalotan language. One day, he meets a Kalotan (heterosexual) couple and their child Kibi. Worf asks Kibi: ‘‘Are you a boy?’’ The kid answers in Kalotan, which of course Worf doesn't understand. Worf turns to the parents (who know English) for explanation. One of them says: "Kibi said: ‘I am a boy.’" The other adds: "Kibi is a girl. Kibi lied." Solve for the sex of Kibi and

the sex of each parent.

We will need some additional predicates specific to this puzzle, and to define the universe of allowed variables values:

Code available in example24.hs

```
-- if a male says something, it must be true
said :: Var -> Predicate -> Predicate
said v p = (v `Is` "male") `implies` p

-- if a male says two things, they must be true
-- if a female says two things, one must be true and one must be false
saidBoth :: Var -> Predicate -> Predicate -> Predicate
saidBoth v p1 p2 = And ((v `Is` "male") `implies` (p1 `And` p2))
                       ((v `Is` "female") `implies` (p1 `orElse` p2))

-- lying is saying something is true when it isn't or saying something isn't true when it is
lied :: Var -> Predicate -> Predicate
lied v p = ((v `said` p) `And` (Not p)) `orElse` ((v `said` (Not p)) `And` p)

-- Test consistency over all allowed settings of the variable.
tryAllValues :: Var -> NDS ()
tryAllValues var = do (setVar var "male") `mplus` (setVar var "female")
                      c <- isConsistent True
                      guard c
```

All that remains to be done is to define the puzzle in the predicate language and get a solution that satisfies all of the predicates:

Code available in example24.hs

```
-- Define the problem, try all of the variable assignments and print a solution.
main :: IO ()
main = do let variables   = []
              constraints = [ Not (Equal "parent1" "parent2"),
                              "parent1" `said` ("child" `said` ("child" `Is` "male")),
                              saidBoth "parent2" ("child" `Is` "female")
                                                 ("child" `lied` ("child" `Is` "male")) ]
              problem     = PS variables constraints
          print $ (`getSolution` problem) $ do tryAllValues "parent1"
                                               tryAllValues "parent2"
                                               tryAllValues "child"
                                               getFinalVars
```

Each call to `tryAllValues` will fork the solution space, assigning the named variable to be `"male"` in one fork and `"female"` in the other. The forks which produce inconsistent variable assignments are eliminated (using the `guard` function). The call to `getFinalVars` applies `guard` again to eliminate inconsistent variable assignments and returns the remaining assignments as the value of the computation.

Prev: Anatomy of a monad transformer | TOC: Contents | Next: Managing the transformer stack

Managing the transformer stack

- Selecting the correct order
- An example with multiple transformers
- Heavy lifting

As the number of monads combined together increases, it becomes increasingly important to manage the stack of monad transformers well.

Selecting the correct order

Once you have decided on the monad features you need, you must choose the correct order in which to apply the monad transformers to achieve the results you want. For instance you may know that you want a combined monad that is an instance of `MonadError` and `MonadState`, but should you apply `StateT` to the `Error` monad or `ErrorT` to the `State` monad?

The decision depends on the exact semantics you want for your combined monad. Applying `StateT` to the `Error` monad gives a state transformer function of type `s -> Error e (a,s)`. Applying `ErrorT` to the `State` monad gives a state transformer function of type `s -> (Error e a,s)`. Which order to choose depends on the role of errors in your computation. If an error means no state could be produced, you would apply `StateT` to `Error`. If an error means no value could be produced, but the state remains valid, then you would apply `ErrorT` to `State`.

Choosing the correct order requires understanding the transformation carried out by each monad transformer, and how that transformation affects the semantics of the combined monad.

An example with multiple transformers

The following example demonstrates the use of multiple monad transformers. The code uses the StateT monad transformer along with the List monad to produce a combined monad for doing stateful nondeterministic computations. In this case, however, we have added the `WriterT` monad transformer to perform logging during the computation. The problem we will apply this monad to is the famous N-queens problem: to place N queens on a chess board so that no queen can attack another.

The first decision is in what order to apply the monad transformers. `StateT s (WriterT w [])` yields a type like: `s -> [((a,s),w)]`. `WriterT w (StateT s [])` yields a type like: `s -> [((a,w),s)]`. In this case, there is little difference between the two orders, so we will choose the second arbitrarily.

Our combined monad is an instance of both `MonadState` and `MonadWriter`, so we can freely mix use of `get`, `put`, and `tell` in our monadic computations.

Code available in example25.hs

```
            Just (b,l) -> do putStr $ show b   -- show the solution
                             putStr $ unlines l -- show the log
            Nothing    -> putStrLn "No solution"
```

```
-- this is the type of our problem description
data NQueensProblem = NQP {board::Board,
                           ranks::[Rank],   files::[File],
                           asc::[Diagonal], desc::[Diagonal]}

-- initial state = empty board, all ranks, files, and diagonals free
initialState = let fileA = map (\r->Pos A r) [1..8]
                   rank8 = map (\f->Pos f 8) [A .. H]
                   rank1 = map (\f->Pos f 1) [A .. H]
                   asc   = map Ascending (nub (fileA ++ rank1))
                   desc  = map Descending (nub (fileA ++ rank8))
               in NQP (Board []) [1..8] [A .. H] asc desc

-- this is our combined monad type for this problem
type NDS a = WriterT [String] (StateT NQueensProblem []) a

-- Get the first solution to the problem, by evaluating the solver computation with
-- an initial problem state and then returning the first solution in the result list,
-- or Nothing if there was no solution.
getSolution :: NDS a -> NQueensProblem -> Maybe (a,[String])
getSolution c i = listToMaybe (evalStateT (runWriterT c) i)

-- add a Queen to the board in a specific position
addQueen :: Position -> NDS ()
addQueen p = do (Board b) <- gets board
                rs <- gets ranks
                fs <- gets files
                as <- gets asc
                ds <- gets desc
                let b'  = (Piece Black Queen, p):b
                    rs' = delete (rank p) rs
                    fs' = delete (file p) fs
                    (a,d) = getDiags p
                    as' = delete a as
                    ds' = delete d ds
                tell ["Added Queen at " ++ (show p)]
                put (NQP (Board b') rs' fs' as' ds')

-- test if a position is in the set of allowed diagonals
inDiags :: Position -> NDS Bool
inDiags p = do let (a,d) = getDiags p
               as <- gets asc
               ds <- gets desc
               return $ (elem a as) && (elem d ds)

-- add a Queen to the board in all allowed positions
addQueens :: NDS ()
addQueens = do rs <- gets ranks
               fs <- gets files
               allowed <- filterM inDiags [Pos f r | f <- fs, r <- rs]
               tell [show (length allowed) ++ " possible choices"]
               msum (map addQueen allowed)

-- Start with an empty chess board and add the requested number of queens,
-- then get the board and print the solution along with the log
main :: IO ()
main = do args <- getArgs
          let n    = read (args!!0)
              cmds = replicate n addQueens
              sol  = (`getSolution` initialState) $ do sequence_ cmds
                                                       gets board
          case sol of
```

The program operates in a similar manner to the previous example, which solved the kalotan puzzle. In this example, however, we do not test for consistency using the `guard` function. Instead, we only create branches that correspond to allowed queen positions. We use the added logging facility to log the number of possible choices at each step and the position in which the queen was placed.

Heavy lifting

There is one subtle problem remaining with our use of multiple monad transformers. Did you notice that all of the computations in the previous example are done in the combined monad, even if they only used features of one monad? The code for these functions in tied unneccessarily to the definition of the combined monad, which decreases their reusability.

This is where the `lift` function from the `MonadTrans` class comes into its own. The `lift` function gives us the ability to write our code in a clear, modular, reusable manner and then lift the computations into the combined monad as needed.

Instead of writing brittle code like:

```
logString :: String -> StateT MyState (WriterT [String] []) Int
logString s = ...
```

we can write clearer, more flexible code like:

```
logString :: (MonadWriter [String] m) => String -> m Int
logString s = ...
```

and then lift the `logString` computation into the combined monad when we use it.

You may need to use the compiler flags `-fglasgow-exts` with GHC or the equivalent flags with your Haskell compiler to use this technique. The issue is that `m` in the constraint above is a type constructor, not a type, and this is not supported in standard Haskell 98.

When using lifting with complex transformer stacks, you may find yourself composing multiple lifts, like `lift . lift . lift $ f x`. This can become hard to follow, and if the transformer stack changes (perhaps you add `ErrorT` into the mix) the lifting may need to be changed all over the code. A good practice to prevent this is to declare helper functions with informative names to do the lifting:

```
liftListToState = lift . lift . lift
```

Then, the code is more informative and if the transformer stack changes, the impact on the lifting code is confined to a small number of these helper functions.

The hardest part about lifting is understanding the semantics of lifting computations, since this depends on the details of the inner monad and the transformers in the stack. As a final task, try to understand the different roles that lifting plays in the following example code. Can you predict what the output of the program will be?

Code available in example26.hs

```
-- this is our combined monad type for this problem
type NDS a = StateT Int (WriterT [String] []) a

{- Here is a computation on lists -}

-- return the digits of a number as a list
getDigits :: Int -> [Int]
getDigits n = let s = (show n)
              in map digitToInt s

{- Here are some computations in MonadWriter -}

-- write a value to a log and return that value
logVal :: (MonadWriter [String] m) => Int -> m Int
logVal n = do tell ["logVal: " ++ (show n)]
              return n

-- do a logging computation and return the length of the log it wrote
getLogLength :: (MonadWriter [[a]] m) => m b -> m Int
getLogLength c = do (_,l) <- listen $ c
                    return (length (concat l))

-- log a string value and return 0
logString :: (MonadWriter [String] m) => String -> m Int
logString s = do tell ["logString: " ++ s]
                 return 0

{- Here is a computation that requires a WriterT [String] [] -}

-- "Fork" the computation and log each list item in a different branch.
logEach :: (Show a) => [a] -> WriterT [String] [] a
logEach xs = do x <- lift xs
                tell ["logEach: " ++ (show x)]
                return x

{- Here is a computation in MonadState -}

-- increment the state by a specified value
addVal :: (MonadState Int m) => Int -> m ()
addVal n = do x <- get
              put (x+n)

{- Here are some computations in the combined monad -}

-- set the state to a given value, and log that value
setVal :: Int -> NDS ()
setVal n = do x <- lift $ logVal n
              put x

-- "Fork" the computation, adding a different digit to the state in each branch.
-- Because setVal is used, the new values are logged as well.
addDigits :: Int -> NDS ()
addDigits n = do x  <- get
                 y <- lift . lift $ getDigits n
                 setVal (x+y)

{- an equivalent construction is:
addDigits :: Int -> NDS ()
addDigits n = do x <- get
                 msum (map (\i->setVal (x+i)) (getDigits n))
-}
```

```
{- This is an example of a helper function that can be used to put all of the lifting logic
   in one location and provide more informative names.  This has the advantage that if the
   transformer stack changes in the future (say, to add ErrorT) the changes to the existing
   lifting logic are confined to a small number of functions.
-}
liftListToNDS :: [a] -> NDS a
liftListToNDS = lift . lift

-- perform a series of computations in the combined monad, lifting computations from other
-- monads as necessary.
main :: IO ()
main = do mapM_ print $ runWriterT $ (`evalStateT` 0) $ do x <- lift $ getLogLength $ logString "hello"
                                                           addDigits x
                                                           x <- lift $ logEach [1,3,5]
                                                           lift $ logVal x
                                                           liftListToNDS $ getDigits 287
```

Once you fully understand how the various lifts in the example work and how lifting promotes code reuse, you are ready for real-world monadic programming. All that is left to do is to hone your skills writing real software. Happy hacking!

Prev: More examples with monad transformers | TOC: Contents | Next: Continuing Exploration

Continuing Exploration

This brings us to the end of this tutorial. If you want to continue learning about the mathematical foundations of monads, there are numerous category theory resources on the internet. For more examples of monads and their applications in the real world, you might want to explore the design of the Parsec monadic parser combinator library and/or the QuickCheck testing tool. You may also be interested in arrows, which are similar to monads but more general.

If you discover any errors — no matter how small — in this document, or if you have suggestions for how it can be improved, please write to the author at jnewbern@yahoo.com.

A physical analogy for monads

Because monads are such abstract entities, it is sometimes useful to think about a physical system that is analogous to a monad instead of thinking about monads directly. In this way, we can use our physical intuition and experiences to gain insights that we can relate back to the abstract world of computational monads.

The particular physical analogy developed here is that of a mechanized assembly line. It is not a perfect fit for monads — especially with some of the higher-order aspects of monadic computation — but I believe it could be helpful to gain the initial understanding of how a monad works.

Begin by thinking about a Haskell program as a conveyor belt. Input goes on end of the conveyor belt and is carried to a succession of work areas. At each work area, some operation is performed on the item on the conveyor belt and the result is carried by the conveyor belt to the next work area. Finally, the conveyor belt carries the final product to the end of the assembly line to be output.

In this assembly line model, our physical monad is a system of machines that controls how successive work areas on the assembly line combine their functionality to create the overall product.

Our monad consists of three parts:

1. Trays that hold work products as they move along the conveyor belt.
2. Loader machines that can put any object into a tray.
3. Combiner machines that can take a tray with an object and produce a tray with a new object. These combiner machines are attached to worker machines that actualy produce the new objects.

We use the monad by setting up our assembly line as a loader machine which puts materials into trays at the beginning of the assembly line. The conveyor belt then carries these trays to each work area, where a combiner machine takes the tray and may decide based on its contents whether to run them through a worker machine, as shown in Figure A-1.

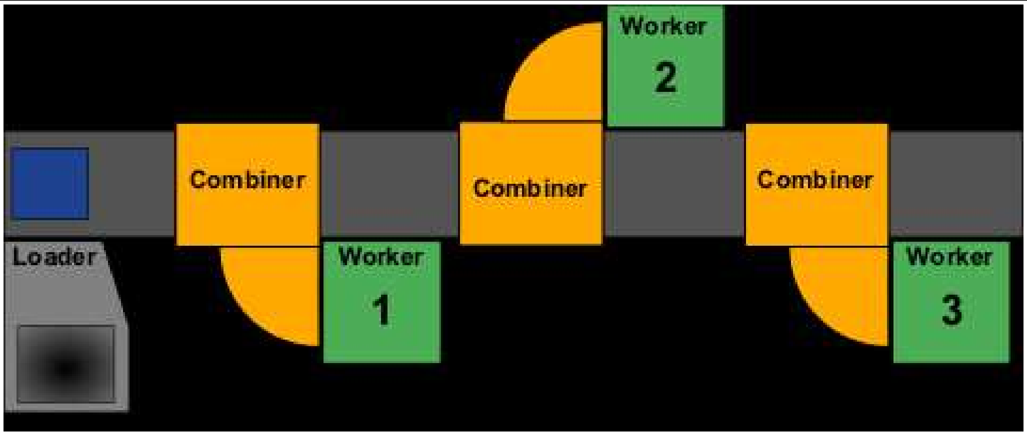

Figure A-1. An assembly line using a monad architecture.

The important thing to notice about the monadic assembly line is that it separates out the work of combining the output of the worker machines from the actual work done by the worker machines. Once they are separated, we can vary them independently. So the same combiner machines could be used on an assembly line to make airplanes and an assembly line to make chopsticks. Likewise, the same worker machines could be used with different combiners to alter the way the final product is produced.

Lets take the example of an assembly line to make chopsticks, and see how it is handled in our physical analogy and how me might represent it as a program in Haskell. We will have three worker machines. The first takes small pieces of wood as input and outputs a tray containing a pair of roughly shaped chopsticks. The second takes a pair of roughly shaped chopsticks and outputs a tray containing a pair of smooth, polished chopsticks with the name of the restaurant printed on them. The third takes a pair of polished chopsticks and outputs a tray containing a finished pair of chopsticks in a printed paper wrapper. We could represent this in Haskell as:

```
-- the basic types we are dealing with
type Wood = ...
type Chopsticks = ...
data Wrapper x = Wrapper x

-- NOTE: the Tray type comes from the Tray monad

-- worker function 1: makes roughly shaped chopsticks
makeChopsticks :: Wood -> Tray Chopsticks
makeChopsticks w = ...

-- worker function 2: polishes chopsticks
polishChopsticks :: Chopsticks -> Tray Chopsticks
polishChopsticks c = ...

-- worker function 3: wraps chopsticks
wrapChopsticks :: Chopsticks -> Tray Wrapper Chopsticks
wrapChopsticks c = ...
```

It is clear that the worker machines contain all of the functionality needed to produce chopsticks. What is missing is the specification of the trays, loader, and combiner machines that collectively make up the Tray monad. Our trays should either be empty or contain a single item. Our loader machine would simply take an item and place it in a tray on the conveyor belt. The combiner machine would take each input tray and pass along empty trays while feeding the contents of non-empty trays to its worker machine. In Haskell, we would define the `Tray` monad as:

```
-- trays are either empty or contain a single item
data Tray x = Empty | Contains x

-- Tray is a monad
instance Monad Tray where
    Empty        >>= _      = Empty
    (Contains x) >>= worker = worker x
    return                  = Contains
    fail _                  = Empty
```

You may recognize the `Tray` monad as a disguised version of the `Maybe` monad that is a standard part of Haskell 98 library.

All that remains is to sequence the worker machines together using the loader and combiner machines to make a complete assembly line, as shown in Figure A-2.

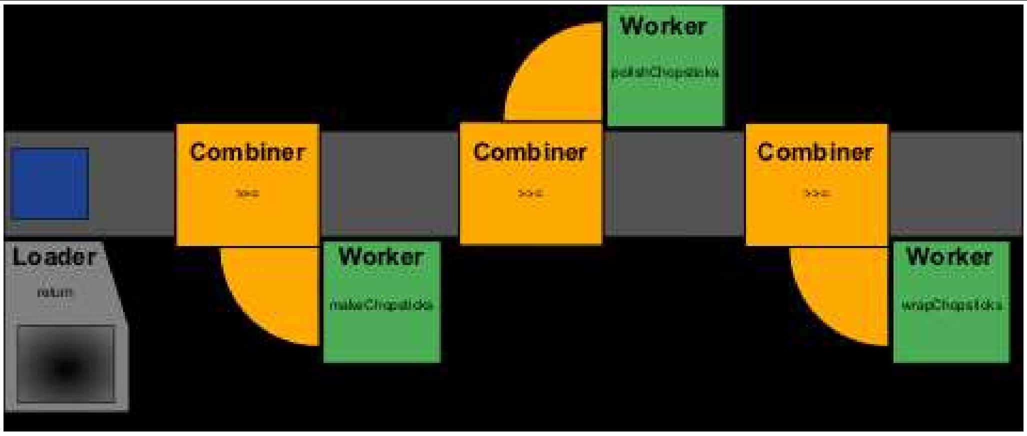

Figure A-2. A complete assembly line for making chopsticks using a monadic approach.

In Haskell, the sequencing can be done using the standard monadic functions:

```
assemblyLine :: Wood -> Tray Wrapped Chopsticks
assemblyLine w = (return w) >>= makeChopsticks >>= polishChopsticks >>= wrapChopsticks
```

or using the built in Haskell "do" notation for monads:

```
assemblyLine :: Wood -> Tray Wrapped Chopsticks
assemblyLine w = do c   <- makeChopsticks w
                    c'  <- polishChopsticks c
                    c'' <- wrapChopsticks c'
                    return c''
```

So far, you have seen how monads are like a framework for building assembly lines, but you probably haven’t been overawed by their utility. To see why we might want to build our assembly line using the monadic approach, consider what would happen if we wanted to change the manufacturing process.

Right now, when a worker machine malfunctions, it uses the `fail` routine to produce an empty tray. The `fail` routine takes an argument describing the failure, but our `Tray` type ignores this and simply produces an empty tray. This empty tray travels down the assembly line and the combiner machines allow it to bypass the remaining worker machines. It eventually reaches the end of the assembly line, where it is brought to you, the quality control engineer. It is your job to figure out which machine failed, but all you have to go on is an empty tray.

You realize that your job would be much easier if you took advantage of the failure messages that are currently ignored by the `fail` routine in your `Tray` monad. Because your assembly line is organized around a monadic approach, it is easy for you to add this functionality to your assembly line without changing your worker machines.

To make the change, you simply create a new tray type that can never be empty. It will always either contain an item or it will contain a failure report describing the exact reason there is no item in the tray.

```
-- tray2s either contain a single item or contain a failure report
data Tray2 x = Contains x | Failed String

-- Tray2 is a monad
instance Monad Tray2 where
     (Failed reason) >>= _      = Failed reason
     (Contains x)    >>= worker = worker x
     return                     = Contains
     fail reason                = Failed reason
```

You may recognize the `Tray2` monad as a disguised version of the `Error` monad that is a standard part of the Haskell 98 libraries.

Replacing the `Tray` monad with the `Tray2` monad instantly upgrades your assembly line. Now when a failure occurs, the tray that is brought to the quality control engineer contains a failure report detailing the exact cause of the failure!

Prev: Standard monad transformers | TOC: Contents | Next: Appendix II - Haskell code examples