Monad

From HaskellWiki

Hint: if you're just looking for an introduction to monads, see Merely monadic or one of the other monad tutorials.

Monad class (base)

```
import Control.Monad (https://hackage.haskell.org/package/base/docs/Control-Monad.html#t%3AMonad)
```

Contents

The `Monad` class

Monads can be viewed as a standard programming interface to various data or control structures, which is captured by Haskell's `Monad` class. All the common monads are members of it:

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

In addition to implementing the class functions, all instances of `Monad` should satisfy the following equations, or *monad laws*:

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

For more information, including an intuitive explanation of why the monad laws should be satisfied, see Monad laws.

As of GHC 7.10, the `Applicative` typeclass is a superclass of `Monad`, and the `Functor` typeclass is a superclass of `Applicative`. This means that all monads are applicatives, all applicatives are functors, and therefore all monads are also functors. For more information, see the Functor hierarchy proposal.

If the `Monad` definitions are preferred, `Functor` and `Applicative` instances can be defined from them with:

```
fmap fab ma  =  do { a <- ma ; return (fab a) }
             --  ma >>= (return . fab)
pure a       =  do { return a }
             --  return a
mfab <*> ma  =  do { fab <- mfab ; a <- ma ; return (fab a) }
             --  mfab >>= (\ fab -> ma >>= (return . fab))
             --  mfab `ap` ma
```

although the recommended order is to define `return` as `pure` if the two would otherwise end up being the same.

Common monads

These include:

- Representing failure using `Maybe` monad
- Nondeterminism using `List` monad to represent carrying multiple values
- State using `State` monad
- Read-only environment using `Reader` monad
- I/O using `IO` monad

`do`-notation

In order to improve the look of code that uses monads, Haskell provides a special form of syntactic sugar called `do`-notation. For example, the following expression:

```
thing1 >>= (\x -> func1 x >>= (\y -> thing2
       >>= (\_ -> func2 y >>= (\z -> return z))))
```

which can be written more clearly by breaking it into several lines and omitting parentheses:

```
thing1  >>= \x ->
func1 x >>= \y ->
thing2  >>= \_ ->
func2 y >>= \z ->
return z
```

can also be written using `do`-notation:

```
do {
  x <- thing1 ;
  y <- func1 x ;
  thing2 ;
  z <- func2 y ;
  return z
  }
```

(the curly braces and the semicolons are optional when the indentation rules are observed).

Code written using `do`-notation is transformed by the compiler to ordinary expressions that use the functions from the `Monad` class (i.e. the two varieties of bind: `(>>=)` and `(>>)`).

When using `do`-notation and a monad like `State` or `IO`, programs in Haskell look very much like programs written in an imperative language as each line contains a statement that can change the simulated global state of the program and optionally binds a (local) variable that can be used by the statements later in the code block.

It is possible to intermix the `do`-notation with regular notation.

More on `do`-notation can be found in a section of Monads as computation and in other tutorials.

Commutative monads

For monads which are *commutative* the order of actions makes no difference (i.e. they *commute*), so the following code:

```
do
  a <- actA
  b <- actB
  m a b
```

is the same as:

```
do
  b <- actB
  a <- actA
  m a b
```

Examples of commutative monads include:

- `Reader` monad
- `Maybe` monad

Monad tutorials

Monads are known for being quite confusing to many people, so there are plenty of tutorials specifically related to monads. Each takes a different approach to monads, and hopefully everyone will find something useful.

See the Monad tutorials timeline for a comprehensive list of monad tutorials.

Monad reference guides

An explanation of the basic `Monad` functions, with examples, can be found in the reference guide A tour of the Haskell Monad functions (https://web.archive.org/web/20201109033750/members.chello.nl/hjgtuyl/tourdemonad.html) by Henk-Jan van Tuyl.

Monad research

A collection of research papers about monads.

Monads in other languages

Implementations of monads in other languages.

- C (http://www.reddit.com/r/programming/comments/1761q/monads_in_c_pt_ii/)
- Clojure (https://github.com/clojure/algo.monads)
- CML.event (http://cml.cs.uchicago.edu/pages/cml.html) ?
- Clean (http://www.st.cs.ru.nl/papers/2010/CleanStdEnvAPI.pdf) State monad
- JavaScript (http://cratylus.freewebspace.com/monads-in-javascript.htm)
- Java (http://www.ccs.neu.edu/home/dherman/browse/code/monads/JavaMonads/)
- Joy (http://permalink.gmane.org/gmane.comp.lang.concatenative/1506)
- LINQ (https://web.archive.org/web/20130522092554/http://research.microsoft.com/en-us/um/people/emeijer/Papers/XLinq%20XML%20Programming%20Refactored%20(The%20Return%20Of%20The%20Monoids).htm)
- Lisp (http://common-lisp.net/project/cl-monad-macros/monad-macros.htm)
- Miranda (http://lambda-the-ultimate.org/node/1136#comment-12448)
- OCaml:
  - OCaml (http://www.cas.mcmaster.ca/~carette/pa_monad/)
  - more (https://mailman.rice.edu/pipermail/metaocaml-users-l/2005-March/000057.html)
  - MetaOcaml (http://www.cas.mcmaster.ca/~carette/metamonads/)
  - A Monad Tutorial for Ocaml (http://blog.enfranchisedmind.com/2007/08/a-monad-tutorial-for-ocaml/)
- Perl6 ? (http://www.reddit.com/r/programming/comments/p66e/are_monads_actually_used_in_anything_except)
- Prolog (http://logic.csci.unt.edu/tarau/research/PapersHTML/monadic.html)
- Python
  - Python (http://code.activestate.com/recipes/439361/)
  - Twisted's Deferred monad (http://www.reddit.com/r/programming/comments/p66e/are_monads_actually_used_in_anything_except/cp8eh)
- Ruby:
  - Ruby (http://moonbase.rydia.net/mental/writings/programming/monads-in-ruby/00introduction.html)
  - and other implementation (http://meta-meta.blogspot.com/2006/12/monads-in-ruby-part-1-identity.html)
- Scheme:
  - Scheme (http://okmij.org/ftp/Scheme/monad-in-Scheme.html)
  - also (http://www.ccs.neu.edu/home/dherman/research/tutorials/monads-for-schemers.txt)
  - Monads & Do notation: Part 1 (https://el-tramo.be/blog/async-monad/) Part 2 (https://el-tramo.be/blog/scheme-monads/)
- Swift (http://www.javiersoto.me/post/106875422394)
- Tcl (http://wiki.tcl.tk/13844)
- The Unix Shell (http://okmij.org/ftp/Computation/monadic-shell.html)
- More monads by Oleg (http://okmij.org/ftp/Computation/monads.html)
- CLL (http://lambda-the-ultimate.org/node/2322): a concurrent language based on a first-order intuitionistic linear logic where all right synchronous connectives are restricted to a monad.

- Collection of links to monad implementations in various languages. (http://lambda-the-ultimate.org/node/1136) on Lambda The Ultimate (http://lambda-the-ultimate.org/).

Unfinished:

- Parsing (http://wiki.tcl.tk/14295), Maybe and Error (http://wiki.tcl.tk/13844) in Tcl

And possibly there exists:

- Standard ML (via modules?)

(If you know of other implementations, please add them here.)

Interesting monads

A list of monads for various evaluation strategies and games:

- Identity monad (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Identity.html) - the trivial monad.
- Optional results from computations (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Maybe.html) - error checking without null.
- Random values (http://hackage.haskell.org/packages/archive/monad-mersenne-random/latest/doc/html/Control-Monad-Mersenne-Random.html) - run code in an environment with access to a stream of random numbers.
- Read only variables (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Reader.html) - guarantee read-only access to values.
- Writable state (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Writer-Lazy.html) - i.e. log to a state buffer
- A supply of unique values (http://www.haskell.org/haskellwiki/New_monads/MonadSupply) - useful for e.g. guids or unique variable names
- ST - memory-only, locally-encapsulated mutable variables (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad-ST.html). Safely embed mutable state inside pure functions.
- Global state (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-State-Lazy.html) - a scoped, mutable state.
- Undoable state effects (http://hackage.haskell.org/packages/archive/Hedi/latest/doc/html/Undo.html) - roll back state changes
- Function application (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad-Instances.html#t:Monad) - chains of function application.
- Functions which may error (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Error.html) - track location and causes of errors.
- Atomic memory transactions (http://hackage.haskell.org/packages/archive/stm/latest/doc/html/Control-Monad-STM.html) - software transactional memory
- Continuations (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Cont.html) - computations which can be interrupted and resumed.
- IO (http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-IO.html#t%3AIO) - unrestricted side effects on the world
- Search monad (http://hackage.haskell.org/packages/archive/level-monad/0.4.1/doc/html/Control-Monad-Levels.html) - bfs and dfs search environments.
- non-determinism (http://hackage.haskell.org/packages/archive/stream-monad/lat

est/doc/html/Control-Monad-Stream.html) - interleave computations with suspension.
- stepwise computation (http://hackage.haskell.org/packages/archive/stepwise/latest/doc/html/Control-Monad-Stepwise.html) - encode non-deterministic choices as stepwise deterministic ones
- Backtracking computations (http://logic.csci.unt.edu/tarau/research/PapersHTML/monadic.html)
- Region allocation effects (http://www.cs.cornell.edu/people/fluet/research/rgn-monad/index.html)
- LogicT (http://hackage.haskell.org/packages/archive/logict/0.5.0.2/doc/html/Control-Monad-Logic.html) - backtracking monad transformer with fair operations and pruning
- concurrent events and threads (http://hackage.haskell.org/packages/archive/monad-task/latest/doc/html/Control-Monad-Task.html) - refactor event and callback heavy programs into straight-line code via co-routines
- QIO (http://hackage.haskell.org/package/QIO) - The Quantum computing monad
- Pi calculus (http://hackage.haskell.org/packages/archive/full-sessions/latest/doc/html/Control-Concurrent-FullSession.html) - a monad for Pi-calculus style concurrent programming
- Commutable monads for parallel programming (http://www-fp.dcs.st-and.ac.uk/~kh/papers/pasco94/subsubsectionstar3_3_2_3.html)
- Simple, Fair and Terminating Backtracking Monad (http://hackage.haskell.org/package/stream-monad)
- Typed exceptions with call traces as a monad (http://hackage.haskell.org/package/control-monad-exception)
- Breadth first list monad (http://hackage.haskell.org/package/control-monad-omega)
- Continuation-based queues as monads (http://hackage.haskell.org/package/control-monad-queue)
- Typed network protocol monad (http://hackage.haskell.org/package/full-sessions)
- Non-Determinism Monad for Level-Wise Search (http://hackage.haskell.org/package/level-monad)
- Transactional state monad (http://hackage.haskell.org/package/monad-tx)
- A constraint programming monad (http://hackage.haskell.org/package/monadiccp)
- A probability distribution monad (http://hackage.haskell.org/package/ProbabilityMonads)
- Sets (http://hackage.haskell.org/package/set-monad) - Set computations
- HTTP (http://hackage.haskell.org/package/http-monad/) - http connections as a monadic environment
- Memoization (http://hackage.haskell.org/package/monad-memo) - add memoization to code

There are many more interesting instances of the monad abstraction out there. Please add them as you come across each species.

Fun

- If you are tired of monads, you can easily get rid of them (http://www.haskell.org.monadtransformer.parallelnetz.de/haskellwiki/Category:Monad).

See also

- What a Monad is not
- Monads as containers
- Monads as computation
- Monad/ST
- Why free monads matter (http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html) (blog article)