

Package ‘monad’

July 23, 2025

Title Operators and Generics for Monads

Version 0.1.1

Description Compose generic monadic function pipelines with `%>>%` and `%>-%` based on implementing the 'S7' generics `fmap()` and `bind()`. Methods are provided for the built-in list type and the maybe class from the 'maybe' package. The concepts are modelled directly after the Monad typeclass in Haskell, but adapted for idiomatic use in R.

License MIT + file LICENSE

URL <https://github.com/mikmart/monad>, <https://mikmart.github.io/monad/>

BugReports <https://github.com/mikmart/monad/issues>

Depends R (>= 4.1)

Imports S7

Suggests maybe, purrr, roxygen2, testthat (>= 3.0.0)

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2.9000

Collate 'pipeop.R' 'monad.R' 'list.R' 'maybe.R' 'monad-package.R'
'utils.R' 'zzz.R'

NeedsCompilation no

Author Mikko Marttila [aut, cre]

Maintainer Mikko Marttila <mikkmart@protonmail.com>

Repository CRAN

Date/Publication 2024-10-28 12:40:03 UTC

Contents

| | |
|------------------------|---|
| functor-laws | 2 |
| List | 2 |
| Maybe | 3 |
| monad | 3 |
| monad-laws | 5 |

Index**7**

| | |
|--------------|---------------------|
| functor-laws | <i>Functor Laws</i> |
|--------------|---------------------|

Description

Classes implementing `fmap()` are expected to satisfy two functor laws: preservation of identity and preservation of composition.

Arguments

| | |
|-------------------|-------------------|
| <code>m</code> | A functor object. |
| <code>f, g</code> | Functions. |

Details

The Haskell functor laws can be translated into R as follows:

Preservation of identity: `m %>>% identity` is equal to `m |> identity()`.

Preservation of composition: `m %>>% (f %.% g)` is equal to `m %>>% g %>>% f`.

Where above `%.%` denotes function composition $\backslash(f, g) \backslash(x) f(g(x))$.

References

https://wiki.haskell.org/Functor#Functor_Laws

See Also

Other implementation laws: [monad-laws](#)

| | |
|------|-----------------------|
| List | <i>The List Monad</i> |
|------|-----------------------|

Description

The `list` built-in type is a monad with element-wise function application as `fmap()` and flattening as `join()`. It follows that `%>>%` is a map operator and `%>-%` is a "flat map" operator. The methods are implemented as wrappers to the `purrr` package.

See Also

`purrr::map()` which implements `fmap()` for `list`.

`purrr::list_flatten()` which implements `join()` for `list`.

Other monads: [Maybe](#)

Examples

```
# The fmap operator corresponds to purrr::map().
list(1, 2) %>>% `+`(1)

# The bind operator is a "flat map" that combines output lists.
list(1, 2) %>-% \(x) list(x * 2, x / 2)
```

Maybe

The Maybe Monad

Description

The **maybe** package implements the Maybe monad. It represents the explicit possibility of absence of a value. Methods for `fmap()`, `bind()` and `join()` are provided for the maybe S3 class as wrappers to functions in the package.

See Also

`maybe::maybe_map()` which implements `fmap()` for maybe.

`maybe::and_then()` which implements `bind()` for maybe.

Other monads: [List](#)

Examples

```
# The fmap operator corresponds to maybe::maybe_map().
maybe::just(1) %>>% `+`(1)
maybe::nothing() %>>% `+`(1)

# The bind operator corresponds to maybe::and_then().
maybe::just(1) %>-% \(x) maybe::just(x + 1)
maybe::just(1) %>-% \(x) maybe::nothing()
maybe::nothing() %>-% \(x) maybe::just(1)
```

monad

Monad Operators and Generics

Description

Classes implementing methods for these S7 generics are called monads. `fmap()` should be implemented such that the [functor laws](#) hold. `bind()` or `join()` should be implemented such that the [monad laws](#) hold. `%>>%` is the `fmap()` pipe operator, and `%>-%` is the `bind()` pipe operator. Operator usage is in the form `m %>>% f(...)`.

Usage

```
lhs %>>% rhs
```

```
lhs %>-% rhs
```

```
fmap(m, f, ...)
```

```
bind(m, f, ...)
```

```
join(m)
```

Arguments

| | |
|---------------------|--|
| <code>m, lhs</code> | A monadic object. |
| <code>f, rhs</code> | A function. For <code>bind()</code> , it should return a monadic object. |
| <code>...</code> | Additional arguments passed to <code>f</code> . |

Value

A monadic object.

Details

Monads are containers for values. `fmap()` transforms the contained value with a function. `bind()` transforms the contained value with a function that returns a monadic object. `join()` takes a monad whose contained value is another monad, and combines them into a new monadic object. It's used to unwrap a layer of monadic structure. Implementing classes typically embed some form of control flow or state management in `bind()` or `join()`.

There's a default implementation for `join()` if you provide `bind()`, and there's a default implementation for `bind()` if you provide `join()` and `fmap()`. For performance reasons you may wish to implement both regardless.

Operators

The pipe operators expect a monadic object as `lhs` and a function or a call expression as `rhs`. A call in `rhs` is treated as partial application of the function `f`. The pipe expression is transformed into a call to the corresponding monad generic with any call arguments in `rhs` passed as additional arguments to `f` in the generic. For example, `m %>>% f(x)` is equivalent to `fmap(m, f, x)` and `m %>-% f(x)` is equivalent to `bind(m, f, x)`.

Trivia

A class that only implements `fmap()` is called a functor.

See Also

The [monad laws](#) and [functor laws](#) that implementations should satisfy. [List](#) and [Maybe](#) for examples of implementing classes.

Examples

```
# We demonstrate by implementing a simple Either monad.
library(S7)

# Start by defining constructors of the Left and Right variants. Conventionally
# a Right variant signifies success and Left an error condition with a context.
left <- function(x) structure(list(value = x), class = c("left", "either"))
right <- function(x) structure(list(value = x), class = c("right", "either"))

# Implement fmap() and bind() methods to gain access to monad operators.
class_either <- new_S3_class("either")

method(fmap, class_either) <- function(m, f, ...) {
  if (inherits(m, "left")) m else right(f(m$value))
}

method(bind, class_either) <- function(m, f, ...) {
  if (inherits(m, "left")) m else f(m$value)
}

# Use with your function that handles errors by returning a monadic value.
mlog <- function(x) {
  if (x > 0) right(log(x)) else left("`x` must be strictly positive.")
}

# fmap() modifies the contained value with a regular function.
mlog(2) %>>% \(x) x + 1
mlog(0) %>>% \(x) x + 1

# bind() modifies the contained value with a function that returns an Either.
mlog(2) %>-% mlog()
mlog(0) %>-% mlog()
```

monad-laws

Monad Laws

Description

Classes implementing `bind()` are expected to satisfy three monad laws: left identity, right identity, and associativity.

Arguments

| | |
|-------------------|---|
| <code>pure</code> | The function to wrap a value in the monad. |
| <code>h, g</code> | Monadic functions. Functions that return monadic objects. |
| <code>a</code> | Any object. |
| <code>m</code> | A monadic object. |

Details

The Haskell monad laws can be translated into R as follows:

Left identity: $\text{pure}(a) \%>-\% h$ is equal to $h(a)$.

Right identity: $m \%>-\% \text{pure}$ is equal to m .

Associativity: $(m \%>-\% g) \%>-\% h$ is equal to $m \%>-\% \lambda(x) g(x) \%>-\% h$.

References

https://wiki.haskell.org/Monad_laws

See Also

Other implementation laws: [functor-laws](#)

Index

* **implementation laws**

 functor-laws, 2

 monad-laws, 5

* **monads**

 List, 2

 Maybe, 3

%>-% (monad), 3

%>% (monad), 3

bind (monad), 3

bind(), 3, 5

fmap (monad), 3

fmap(), 2, 3

functor laws, 3, 4

functor-laws, 2

join (monad), 3

join(), 2, 3

List, 2, 3, 4

Maybe, 2, 3, 4

maybe::and_then(), 3

maybe::maybe_map(), 3

monad, 3

monad laws, 3, 4

monad-laws, 5

purrr::list_flatten(), 2

purrr::map(), 2