monad

Build composable Typst DSLs with lawful sequencing.

Version 0.0.1

This manual is generated with tidy. Every module of the library is rendered below with parameter types, return types, and runnable code examples. Pair this with the `README.md` for design rationale and the `examples/` directory for end-to-end programs.

Contents

Introduction

The `monad` package gives you the building blocks for designing monadic DSLs in Typst — the State-flavored "builder" pattern that several Typst packages already use, but with proper monad structure (`pure`, `bind`, the three monad laws) so your DSLs compose predictably.

There are three usage idioms:

1. **Block-joined builder** (`free.make`) — define your vocabulary as a dict of handlers; users write programs like `SomeEnv({ Op1; Op2 })`.
2. **Sequenced with named results** (`do-bind` + `let-bind`) — when later steps depend on earlier **values**, not just shared state. Works for every instance.
3. **Define your own instance** — a monad is just a dict `(pure: .., bind: ..)`. Build one; verify it with `check-laws`.

The next chapter (*A guided tour*) walks through the abstraction from scratch — recommended on a first read. The remaining chapters are auto-generated reference documentation for every public symbol.

A guided tour

This chapter derives the State monad from scratch — not as an abstract definition, but as the natural conclusion of trying to write some ordinary stateful code cleanly. Once we have it, we look at how this package packages the result, and at how the same recipe applies to other useful effects.

If you have never used monads before, read this in order. If you have, skip to the last two sections.

The problem: threading state by hand

Suppose we want to compute a counter through a small pipeline of operations:

```
#let s0 = (count: 0)
#let s1 = (count: s0.count + 5)
#let s2 = (count: s1.count + 10)
#let s3 = (count: s2.count * 2)
// s3 = (count: 30)
```

It works. But every line names the state we just produced so the next line can read from it. The names are bookkeeping — we don't actually care that the intermediate values existed.

Worse, if any line forgets to thread `s2` correctly and reads `s1` instead, we get a silent bug. The cost of the bookkeeping is real.

Each step as a function

The first move is to make the steps reusable. Lift "add `n`" into a function from a state to a new state:

```
#let add(n) = state => (count: state.count + n)
#let double = state => (count: state.count * 2)
```

Now composition is function application:

```
#double(add(10)(add(5)((count: 0))))
// (count: 30)
```

This reads inside-out and is awkward to scale, but at least the intermediate names have evaporated.

Each step returns a value too

Often a step computes something the **next** step needs — not just a state, but a value. So let every step return **both**: a new state and a value.

```
#let add(n) = state => {
  let next = state.count + n
  ((count: next), next)
}
```

Now the previous bookkeeping comes back, but for a reason — the next step actually uses `v`:

```
#let prog = state => {
  let (s1, v1) = add(5)(state)
  let (s2, v2) = add(10)(s1)
  let (s3, v3) = ((count: s2.count * v2), s2.count * v2)
  (s3, v3)
}
#prog((count: 0))
// ((count: 150), 150)
```

Every line does the same dance: unpack `(state, value)`, feed `state` to the next call, possibly use the value. That dance is mechanical.

Factoring out the threading: `bind`

Whenever the same five-line dance shows up in every program, give it a name and write it once.

```
#let bind(m, k) = state => {
  let (s2, v) = m(state)
  k(v)(s2)
}
```

Read it: "Run action `m` to get a new state and a value. Pass the value to the callback `k`. `k` returns **another** action. Run it on the new state. Return its result."

Now the pipeline:

```
#let prog = bind(add(5), v1 =>
  bind(add(10), v2 =>
    state => ((count: state.count * v2), state.count * v2)))
```

Still nested — we'll address that — but the threading is gone. Every intermediate `(s, v)` is hidden inside `bind`.

Lifting plain values: `pure`

What if a step doesn't need to touch the state at all — it just wants to return a plain value? We need to wrap it into the same `state -> (state, value)` shape.

```
#let pure(x) = state => (state, x)
```

That's all `pure` does. It takes a value and produces a no-op action that yields that value.

That's it. That's the monad.

Two functions:

```
#let pure(x) = state => (state, x)
#let bind(m, k) = state => {
  let (s2, v) = m(state)
  k(v)(s2)
}
```

Plus a convention — actions have shape `state -> (state, value)` — and you have the State monad. Every other combinator (sequence two actions, discard a value, repeat an action, fmap a plain function over the inner value) can be derived from `pure` and `bind` alone. That's why they're the foundation.

Why "monad"?

The name is from category theory and the technical definition is unimportant for using the package. The thing that matters is the **contract** — `pure` and `bind` must satisfy three laws:

1. **Left identity**. `bind(pure(a), f) == f(a)`. If you lift a plain value and then immediately bind, you should get the same result as if you had just called `f(a)` directly. Lifting then binding is a no-op.
2. **Right identity**. `bind(m, pure) == m`. If you bind an action with `pure` as the callback, nothing changes. `pure` is the identity of `bind`.
3. **Associativity**. `bind(bind(m, f), g) == bind(m, x => bind(f(x), g))`. How you parenthesize a chain of binds doesn't matter; what matters is the order of the actions.

These three laws are what make every derived combinator behave predictably. A "monad" that doesn't satisfy them is a buggy monad — combinators built on top will not do what they claim.

The package's `laws.check-laws` will test a candidate instance against the laws by sampling actions. We will use it later.

Reading nested binds cleanly: `do-bind`

The biggest practical wart of `bind` is that long pipelines turn into trees of nested callbacks. The package gives you a flattener, `do-bind`, plus a small marker `let-bind` that tags steps which want to capture the previous value:

```
#do-bind(state.monad, (
  state.put-at("x", 5),
  state.get-at("x"),
  let-bind(v => state.put-at("y", v * 2)),
  state.get-at("y"),
))
```

This is exactly the nested `bind` you would write by hand, but linear. Read top to bottom; treat each `let-bind(v => ...)` as "the previous value is bound to `v`".

Block-join sugar with `free.make`

When every step is a **named command** over a shared state and you don't need to capture per-step values (just chain commands), there's even less ceremony. Define the commands as handlers and let `free.make` package the rest:

```
#let builder = free.make(handlers: (
  Set: (key, val) => state => {
    let s = state
    s.insert(key, val)
    (s, val)
  },
  Add: (key, n) => state => {
    let cur = state.at(key, default: 0)
    let s = state
    s.insert(key, cur + n)
    (s, cur + n)
  },
))
#let Set = builder.ops.Set
#let Add = builder.ops.Add
#let eval = builder.eval

#eval({ Set("x", 2); Add("x", 3) })
```

```
(state: (x: 5), value: 5)
```

Each constructor wraps its action in a 1-tuple. Typst joins arrays via `+`, so `{ Set(..); Add(..) }` produces `(set-action, add-action)` which `eval` interprets in order.

`if` and `for` work inside the block too:

```
#let builder = free.make(handlers: (
  Add: n => state => {
    let cur = state.at("count", default: 0)
    let s = state; s.insert("count", cur + n)
    (s, cur + n)
  },
))
#let Add = builder.ops.Add
#let eval = builder.eval

#eval({
  for n in (1, 2, 3, 4) { Add(n) }
  if true { Add(100) }
})
```

```
(state: (count: 110), value: 110)
```

Other shapes, same protocol

The recipe — define `pure` and `bind` over some shape, check the laws, get every derived combinator for free — is not specific to State. The package ships five other instances. The shape of "an action" changes; the `pure` / `bind` contract does not.

Option — `M a = some(a) | nothing`. Models a computation that may produce no value (Rust's `Option<T>`). `bind` short-circuits on `nothing`:

```
#let safe-div(a, b) = if b == 0 { option.nothing } else
{ option.some(a / b) }

#do-bind(option.monad, (
  safe-div(100, 5),
  let-bind(x => safe-div(x, 2)),
  let-bind(y => pure(option.monad, y + 1)),
))
```

```
(tag: "Some", value: 11.0)
```

Replace `0` with a non-zero divisor anywhere and the chain succeeds. Try dividing by zero and the rest of the chain is skipped:

```
#let safe-div(a, b) = if b == 0 { option.nothing } else
{ option.some(a / b) }

#do-bind(option.monad, (
  safe-div(100, 5),
  let-bind(x => safe-div(x, 0)),
  let-bind(y => pure(option.monad, y + 1)),
))
```

```
(tag: "None")
```

Result — `M a = ok(a) | err(e)`. Like Option but the failure value is typed. `bind` short-circuits on `err`, threading the error through.

Reader — `M a = env -> a`. A function from an environment. Useful for dependency injection: pass config once at the top, read it with `ask` anywhere in the chain.

Writer — `M a = (log, a)`. Accumulates a log alongside the value. The log type must form a monoid (an identity and an associative combine); the default is array concatenation.

Identity — `M a = a`. The trivial monad. Useful for testing the abstract combinators and as the no-effect baseline.

Defining your own

A monad in this package is just a dict with `pure` and `bind` fields. You can build one in a few lines, then use `check-laws` to confirm it actually satisfies the contract. Most of your custom monads will look like one of the six above with a small twist — say, State with an extra error channel, or Writer with a custom log monoid.

```
#let my-monad = (
  pure: x => ...,
  bind: (m, k) => ...,
)

#let report = check-laws(my-monad, my-eq, (
  values: (...),
  actions: (...),
  "arrows-f": (...),
  "arrows-g": (...),
))
#assert(report.passed, message: repr(report.failures))
```

The `examples/custom-monad.typ` file in this repository walks through defining an **incorrect** monad — one that counts bind invocations — and shows how `check-laws` catches the associativity violation. It is worth reading once.

What we covered

- Stateful sequencing motivates `bind` and `pure`.
- A monad is `(pure, bind)` satisfying three laws.
- Reading nested binds is helped by `do-bind` + `let-bind`.
- Sequencing named commands is helped by `free.make`.
- The same protocol covers Maybe, Result, Reader, Writer, and your own custom monads.

In practice

This chapter shows a worked use case for each instance the package ships. The point is not exhaustive coverage but a concrete answer to the question "when would I reach for this?" — picking the wrong monad for a problem is the single most common cause of awkward code that "uses monads" but does not benefit from them.

The following examples are deliberately small and runnable.

Identity — swappable no-effect baseline

The Identity monad on its own does nothing useful. Its job is to be the **null effect** in code that is generic over a monad instance. Write a combinator once for an arbitrary monad; pass `identity.monad` when you do not want effects, swap in `result.monad` or `state.monad` later when you do.

A small example: a transformation pipeline parameterised by its effect. With Identity, the pipeline is plain function composition.

```
#let pipeline(monad, steps, x) = {
  let acc = pure(monad, x)
  for f in steps {
    acc = bind(monad, acc, f)
  }
  acc
}
#pipeline(identity.monad, (
  x => x + 1,
  x => x * 2,
  x => x - 3,
), 10)
```

```
19
```

Swap the monad and the same `pipeline` definition handles error short-circuiting, accumulated logs, or threaded state without any change to the calling code:

```
#pipeline(result.monad, (
  x => result.ok(x + 1),
  x => if x > 100 { result.err("overflow") } else
{ result.ok(x * 2) },
  x => result.ok(x - 3),
), 10)
```

```
(tag: "ok", value: 19)
```

Option — chained lookups where any step may be absent

Reach for Option when you have a sequence of lookups, each of which could yield "nothing here". The classic shape: lookup a thing, then look up something derived from it, then derive again. Without a monad, this becomes a tower of `if-some` checks; with `do-bind`, the chain reads top-to-bottom and short-circuits the moment any step is absent.

```
#let users = (
  alice: (org: "acme"),
  bob: (org: "globex"),
)
#let orgs = (
  acme: (plan: "pro"),
  globex: (plan: "free"),
)
#let lookup(dict, key) = if key in dict {
  option.some(dict.at(key))
} else {
  option.nothing
}
#do-bind(option.monad, (
  lookup(users, "alice"),
  let-bind(user => lookup(orgs, user.org)),
  let-bind(org => pure(option.monad, org.plan)),
))
```

```
(tag: "Some", value: "pro")
```

If any link in the chain is missing, the rest of the chain does not run and the whole expression is `nothing`:

```
#do-bind(option.monad, (
  lookup(users, "carol"),
  let-bind(user => lookup(orgs, user.org)),
  let-bind(org => pure(option.monad, org.plan)),
))
```

```
(tag: "None")
```

Result — validation pipeline with typed errors

Result is Option with a typed failure value. Reach for it when the **reason** a step failed matters — when "nothing" is not enough and the caller needs to know what went wrong. The natural use case is input validation: each step parses or checks the input; the first failure short-circuits and the error survives to the caller.

```
#let parse-int-strict(s) = {
  if type(s) != str {
    result.err("expected a string, got " +
type(s))
  } else if s.contains(regex("[^0-9]")) {
    result.err("not a valid integer: " + s)
  } else {
    let n = int(s)
    result.ok(n)
  }
}
#let must-be-positive(n) = if n > 0 {
  result.ok(n)
} else {
  result.err("must be positive, got " + str(n))
}
#let must-be-small(n) = if n < 100 {
  result.ok(n)
} else {
  result.err(
    "must be less than 100, got " + str(n)
  )
}
#let parse(s) = do-bind(
  result.monad, (
    parse-int-strict(s),
    let-bind(must-be-positive),
    let-bind(must-be-small),
))
#parse("42")
```

```
(tag: "ok", value: 42)
```

A bad input fails at the first violated rule and carries the message back unchanged through every subsequent step:

```
#parse("not a number")

#parse("250")
```

```
(
  tag: "err",
 value: "not a valid integer: not a number",
)
(
  tag: "err",
  value: "must be less than 100, got 250",
)
```

State — threading a counter through a build

State is the right tool whenever many disparate steps need to read or update some shared "context" — figure numbers, section labels, a list of registered citations, a running cost or weight. The state dict is arbitrary; you decide what lives in it.

A common Typst use is automatic numbering. The action `next-figure` increments the counter under `"counter"` and returns the new value. Subsequent actions read the value and use it.

```
#let next-figure = state.modify-at("counter", n => n + 1,
default: 0)

#let captions = do-bind(state.monad, (
  next-figure,
  let-bind(n => state.put-at(
    "intro",
    "Figure " + str(n) + ": Overview",
  )),
  next-figure,
  let-bind(n => state.put-at(
    "data",
    "Figure " + str(n) + ": Data",
  )),
  next-figure,
  let-bind(n => state.put-at(
    "summary",
    "Figure " + str(n) + ": Summary",
  )),
))

#state.run(captions, (:))
```

```
(
  (
    counter: 3,
    intro: "Figure 1: Overview",
    data: "Figure 2: Data",
    summary: "Figure 3: Summary",
  ),
  "Figure 3: Summary",
)
```

The runs are reproducible: feed the same initial state and you get the same final state, every time. Nothing inside `captions` is mutable in the usual sense — `state.put-at` returns a **new** state and `bind` threads it through.

Reader — dependency injection through a config

Reader is for "I need this configuration value, but I don't want to plumb it through every function call". Pass the environment once at the top via `reader.run`; read it inside any action via `reader.ask` or `reader.ask-at`. The environment is read-only — every action sees the same value unless you scope a local override with `reader.local`.

The classic use is per-theme rendering. The same `render-heading` action gives different output depending on which theme is supplied at the call site.

```
#let render-heading(text) = do-bind(reader.monad, (
  reader.ask-at("theme"),
  let-bind(theme => pure(reader.monad,
    if theme == "dark" {
      "[DARK] " + text
    } else if theme == "print" {
      "« " + text + " »"
    } else {
      "** " + text + " **"
    })),
))

#reader.run(
  render-heading("Introduction"),
  (theme: "dark"),
)

#reader.run(
  render-heading("Introduction"),
  (theme: "print"),
)
```

```
[DARK] Introduction
« Introduction »
```

Writer — collecting warnings while validating

Writer accumulates a log alongside the value. Reach for it when a computation needs to emit auxiliary facts as it runs: warnings, audit records, citation entries, debug traces. The log type must form a monoid; the default instance uses arrays under concatenation, so `tell` appends to a list of records.

A natural fit is non-fatal validation — checks that should record a warning rather than abort. The result is `(warnings, value)`.

```
#let validate-field(name, value) = {
  if value == none {
    bind(writer.monad,
      writer.tell(("missing field: " + name,)),
      _ => pure(writer.monad, none))
  } else {
    pure(writer.monad, value)
  }
}

#let validate-form(data) = sequence(
  writer.monad, (
    validate-field(
      "name",
      data.at("name", default: none),
    ),
    validate-field(
      "email",
      data.at("email", default: none),
    ),
    validate-field(
      "age",
      data.at("age", default: none),
    ),
))

#validate-form((
  name: "Alice",
  email: none,
  age: 30,
))
```

```
(
  ("missing field: email",),
  ("Alice", none, 30)
)
```

A clean form yields an empty warning list and the final value:

```
#validate-form((
  name: "Alice",
  email: "a@b.c",
  age: 30,
))
```

```
((), ("Alice", "a@b.c", 30))
```

Picking the right monad

The instances answer different shapes of question. A quick guide:

Use	When you need
Identity	a no-effect baseline for monad-generic code
Option	chained lookups that may produce no value
Result	chained validation/parsing with typed failure reasons
State	a counter, accumulator, or evolving context shared across steps
Reader	config or dependencies passed once, read anywhere
Writer	non-fatal warnings, audit logs, or other side records

When more than one fits, you can either pick the simplest one that covers your needs, or combine effects by writing a custom instance. `check-laws` will tell you whether your hybrid is still a lawful monad.

The rest of this manual is reference: every public function with its parameter list, return type, and a runnable example.

Core combinators

The combinators in `src/core.typ` operate over an opaque **monad instance**: a dict with at minimum `pure: a -> M a` and `bind: (M a, a -> M b) -> M b`. Optional fields `fmap`, `join`, and `ap` are derived from `bind` if absent.

core

ap

Applicative application: run a monadic function and a monadic argument, then apply.

```
#ap(option.monad, option.some(x => x + 1),
option.some(4))
```

```
(tag: "Some", value: 5)
```

Parameters

```
ap(
  monad: dictionary,
  mf: any,
  ma: any
) -> any
```

mf `any`

Function-valued monadic.

ma `any`

Argument-valued monadic.

bind

Sequence two computations, feeding the result of the first into a callback that produces the second. The bedrock combinator of every monad; everything else in this file is derived from `pure` and `bind`.

```
#bind(option.monad, option.some(3), x =>
option.some(x + 1))
```

```
(tag: "Some", value: 4)
```

Parameters

```
bind(
  monad: dictionary,
  m: any,
  k: function
) -> any
```

m `any`

First monadic value.

k `function`

Callback `value -> M b`.

do-bind

Fold a flat sequence of actions and callbacks via `bind()`. Each step is either a monadic action (its value is discarded) or `let-bind(v => ...)` (the previous value is bound to `v`).

Reads like Haskell `do`: write actions and `let-bind`-wrapped arrows in the order they should fire.

```
#let prog = do-bind(state.monad, (
  state.put-at("x", 10),
  state.get-at("x"),
  let-bind(v => state.put-at("y", v * 2)),
))
#state.run(prog, (:))
```

```
((x: 10, y: 20), 20)
```

Parameters

```
do-bind(
  monad: dictionary,
  steps: array
) -> any
```

steps `array`

Steps in execution order. First must be an action, not a let-bind.

fmap

Apply a plain function to the value inside a monadic context, without flattening. Equivalent to `bind(m, x => pure(f(x)))`; instances may provide a faster implementation via the optional `fmap` field.

```
#fmap(option.monad, x => x * 10,
option.some(4))
```

```
(tag: "Some", value: 40)
```

Parameters

```
fmap(
  monad: dictionary,
  f: function,
  m: any
) -> any
```

f `function`

Plain transformation.

m `any`

Source monadic value.

for-m

Like `map-m()` with the array and function arguments swapped — convenient when the function is a multi-line closure.

Parameters

```
for-m(
  monad: dictionary,
  xs: array,
  f: function
) -> any
```

join

Collapse a nested monadic value `M (M a)` into `M a`. Useful when a callback produces an already-wrapped value and you want to unwrap a layer.

```
#join(option.monad,
option.some(option.some(5)))
```

```
(tag: "Some", value: 5)
```

Parameters

```
join(
  monad: dictionary,
  mm: any
) -> any
```

mm `any`

Doubly-wrapped value.

kleisli

Compose a list of Kleisli arrows (`a -> M b`, `b -> M c`, ...) into a single `a -> M z` arrow.

```
#let inc = x => option.some(x + 1)
#let double = x => option.some(x * 2)
#(kleisli(option.monad, (inc, double)))(3)
```

```
(tag: "Some", value: 8)
```

Parameters

```
kleisli(
  monad: dictionary,
  fs: array
) -> function
```

fs `array`

Array of Kleisli arrows.

let-bind

Wrap a callback so `do-bind()` can tell it apart from a plain action. Required because in State/Reader/Writer monads, **actions** are themselves functions — `type()` alone cannot disambiguate.

```
#let-bind(v => option.some(v + 1))
```

```
(_do-bind-cont: (..) => ..)
```

Parameters

```
let-bind(k: function) -> dictionary
```

k `function`

value -> M b.

map-m

Map a monad-producing function across an array, then sequence the results. `mapM` from Haskell.

```
#map-m(option.monad, x => option.some(x * 2),
(1, 2, 3))
```

```
(tag: "Some", value: (2, 4, 6))
```

Parameters

```
map-m(
  monad: dictionary,
  f: function,
  xs: array
) -> any
```

f `function`

a -> M b.

xs `array`

Array of inputs.

pure

Lift a plain value into a monadic context.

```
#pure(option.monad, 7)
```

```
(tag: "Some", value: 7)
```

Parameters

```
pure(
  monad: dictionary,
  x: any
) -> any
```

monad `dictionary`

The monad instance dict (with `pure` and `bind` fields).

x `any`

The value to inject.

replicate

Repeat an action `n` times and collect every produced value into an array.

```
#replicate(option.monad, 3, option.some("hi"))
```

```
(tag: "Some", value: ("hi", "hi",
"hi"))
```

Parameters

```
replicate(
  monad: dictionary,
  n: int,
  m: any
) -> any
```

n `int`

Number of repetitions (zero or negative produces `pure(())`).

seq

Sequence a list of monadic values, discarding intermediate results. Returns the final value. Accepts nested arrays (from Typst block-join) and flattens them before folding.

```
#seq(option.monad, (option.some(1),
option.some(2), option.some(3)))
```

```
(tag: "Some", value: 3)
```

Parameters

```
seq(
  monad: dictionary,
  ms: array
) -> any
```

ms `array`

Array (possibly nested) of monadic values.

sequence

Sequence a list of monadic values, **collecting** every intermediate result into an array. The returned action yields a tuple of length `ms.len()`.

```
#state.run(
  sequence(state.monad, (state.put-at("a", 1),
state.put-at("b", 2))),
  (:),
)
```

```
((a: 1, b: 2), (1, 2))
```

Parameters

```
sequence(
  monad: dictionary,
  ms: array
) -> any
```

ms `array`

Array of monadic values.

unless

Inverse of `when()` — run the action only when the condition is `false`.

Parameters

```
unless(
  monad: dictionary,
  cond: bool,
  m: any
) -> any
```

void

Discard the value produced by an action while keeping its effect.

Parameters

```
void(
  monad: dictionary,
  m: any
) -> any
```

when

Run an action only if a condition holds, otherwise produce `pure(none)`.

```
#when(option.monad, 5 > 3, option.some("yes"))
```

```
(tag: "Some", value: "yes")
```

Parameters

```
when(
  monad: dictionary,
  cond: bool,
  m: any
) -> any
```

do `any`

Alias for `seq()`. Reads more naturally inside builder blocks: `do(state.monad, { ... })`.

Monad-law verifiers

Runtime checkers for the three monad laws (`bind`/`pure` interactions) and the two functor laws (`fmap` interactions). Equality on monadic values is supplied as a predicate — use `state-eq` / `reader-eq` for monads whose values are closures.

laws

check-associativity

Third monad law: bind is associative. `bind(bind(m, f), g) == bind(m, x => bind(f(x), g))`.

Parameters

```
check-associativity(
  monad: dictionary,
  eq: function,
  m: any,
  f: function,
  g: function
) -> bool
```

f `function`

`a -> M b`.

g `function`

`b -> M c`.

check-fmap-compose

Functor composition law: `fmap(g ∘ f, m) == fmap(g, fmap(f, m))`.

Parameters

```
check-fmap-compose(
  monad: dictionary,
  eq: function,
  m: any,
  f: function,
  g: function
) -> bool
```

check-fmap-identity

Functor identity law: `fmap(id, m) == m`.

Parameters

```
check-fmap-identity(
  monad: dictionary,
  eq: function,
  m: any
) -> bool
```

check-laws

Run every law across the cross-product of supplied sample arrays.

`samples` is a dict with optional keys:

- `values`: array of plain values `a`
- `actions`: array of monadic values `M a`
- `arrows-f`: array of `a -> M b`
- `arrows-g`: array of `b -> M c`
- `plain-fs`: array of `a -> b`
- `plain-gs`: array of `b -> c`

Returns `(passed: bool, failures: array)`.

```
#let report = check-laws(option.monad, (a, b)
=> a == b, (
  values: (1, 2),
  actions: (option.some(5), option.nothing),
  "arrows-f": (x => option.some(x + 1),),
  "arrows-g": (x => option.some(x * 2),),
))
#report.passed
```

```
true
```

Parameters

```
check-laws(
  monad: dictionary,
  eq: function,
  samples: dictionary
) -> dictionary
```

check-left-identity

First monad law: lifting then binding equals direct application. `bind(pure(a), f) == f(a)`.

```
#check-left-identity(option.monad, (a, b) => a
== b, 5, x => option.some(x + 1))
```

```
true
```

Parameters

```
check-left-identity(
  monad: dictionary,
  eq: function,
  a: any,
  f: function
) -> bool
```

eq `function`

Equality predicate on monadic values.

a `any`

Sample value to pump through `pure`.

f `function`

Kleisli arrow `a -> M b`.

check-right-identity

Second monad law: binding `pure` as the continuation is a no-op. `bind(m, pure) == m`.

Parameters

```
check-right-identity(
  monad: dictionary,
  eq: function,
  m: any
) -> bool
```

reader-eq

Equality helper for Reader-shaped monads. Same as `state-eq()` but observes across sample environments.

Parameters

```
reader-eq(
  run: function,
  sample-envs: array
) -> function
```

run `function`

Reader `run` function.

state-eq

Equality helper for State-shaped monads. Returns a predicate that observes two state actions by running them across the supplied initial states and comparing outputs.

```
#let eq = state-eq(state.run, ((:), (x: 1),
(x: 5, y: 10)))
```

Parameters

```
state-eq(
  run: function,
  sample-states: array
) -> function
```

run `function`

State `run` function.

sample-states `array`

Initial states to sample.

Free / operational builder

`free.make` turns a dict of named state handlers into a State-backed DSL. The resulting bundle exposes constructors (returning 1-tuples so block-join sugar works), an interpreter, and the underlying State monad instance.

free

lift

Lift a hand-written State action into the builder action shape (a 1-tuple). Use this when mixing custom State combinators with the constructors returned by `make()`.

```
#lift(state.get-at("x"))
```

```
((..) => ..,)
```

Parameters

```
lift(action: function) -> array
```

action `function`

Bare State action `state -> (state, value)`.

make

Build a State-backed DSL from a dictionary of named handlers.

Each handler has shape `(..args) -> state -> (state', value)` — that is, it returns a State action when called with its DSL-level arguments. The returned bundle exposes:

- `.ops` — a dict of **constructors**. Each wraps its action in a 1-tuple so that block-join sugar `{ Op1(..); Op2(..) }` concatenates actions.
- `.monad` — the underlying State monad instance for use with `bind` etc.
- `.eval(body)` — interpret a body, returning `(state:, value:)`.
- `.eval-with(body, start:)` — like `eval` but with a custom initial state.
- `.eval-state(body)` / `.eval-value(body)` — partial views.
- `.init` — the default initial state.

```
#let builder = free.make(handlers: (
  Set: (k, v) => s => { let s2 = s;
s2.insert(k, v); (s2, v) },
  Add: (k, n) => s => {
    let cur = s.at(k, default: 0)
    let s2 = s; s2.insert(k, cur + n); (s2,
cur + n)
  },
))
#let Set = builder.ops.Set
#let Add = builder.ops.Add
#let eval = builder.eval
#eval({ Set("x", 2); Add("x", 3) }).value
```

```
5
```

Parameters

```
make(
  handlers: dictionary,
  init: dictionary
) -> dictionary
```

handlers `dictionary`

Dict mapping op name to handler.

Default: `(:)`

init `dictionary`

Default initial state passed to `.eval`.

Default: `(:)`

Identity instance

The trivial monad: `M a = a`. Useful as a baseline and for tests.

identity

run

Extract the value from an Identity action. Trivial — it is the action itself — but provided for API symmetry with the other instances.

Parameters

```
run(m: any) -> any
```

monad `dictionary`

Identity monad instance. `pure` is the identity function, `bind` is reverse application. `M a` is just `a`.

```
#pure(identity.monad, 7)
#bind(identity.monad, 3, x => x + 1)
```

```
7 4
```

Option instance

`M a = some(a) | nothing`. Models computations that may produce no value (Rust's `Option<T>`). `bind` short-circuits on `nothing`: once a chain produces `nothing`, every subsequent step is skipped.

option

is-none

Test whether an Option is `None`.

Parameters

```
is-none(m: dictionary) -> bool
```

is-some

Test whether an Option is `Some`.

Parameters

```
is-some(m: dictionary) -> bool
```

map-or

Eliminator: apply `f` to the inner value, or fall back to `default`. Mirrors Rust's `Option::map_or`.

Parameters

```
map-or(
  default: any,
  f: function,
  m: dictionary
) -> any
```

run

Identity `run` for API symmetry — Option values are already concrete.

Parameters

```
run(m: dictionary) -> dictionary
```

some

Wrap a value as `Some`.

```
#some(42)
```

```
(tag: "Some", value: 42)
```

Parameters

```
some(x: any) -> dictionary
```

unwrap-or

Extract the inner value or return a default for `None`.

```
#unwrap-or(0, option.some(7))
#unwrap-or(0, option.nothing)
```

```
0 0
```

Parameters

```
unwrap-or(
  default: any,
  m: dictionary
) -> any
```

nothing `dictionary`

The `None` sentinel — named `nothing` because `none` is a reserved Typst literal and cannot be used as an identifier. Distinguishable from Typst's `none` because it is a tagged dict.

monad `dictionary`

Option monad instance. `bind` short-circuits on `None`: once a chain produces `None`, every subsequent step is skipped.

```
#bind(option.monad, option.some(3), x =>
option.some(x * 2))
#bind(option.monad, option.nothing, x =>
option.some(x * 2))
```

```
(tag: "Some", value: 6) (tag: "None")
```

Result instance

`M a = ok(a) | err(e)`. Models computations that may fail with a typed error value, threaded through `bind` unchanged on the error path.

result

err

Wrap an error value as `err`.

```
#err("not found")
```

```
(tag: "err", value: "not found")
```

Parameters

```
err(e: any) -> dictionary
```

is-err

Test whether a Result is `err`.

Parameters

```
is-err(m: dictionary) -> bool
```

is-ok

Test whether a Result is `ok`.

Parameters

```
is-ok(m: dictionary) -> bool
```

map-err

Transform the error value, leaving `ok` untouched.

```
#map-err(s => "ERR: " + s, result.err("boom"))
```

```
(tag: "err", value: "ERR: boom")
```

Parameters

```
map-err(
  f: function,
  m: dictionary
) -> dictionary
```

ok

Wrap a success value as `ok`.

```
#ok(42)
```

```
(tag: "ok", value: 42)
```

Parameters

```
ok(x: any) -> dictionary
```

run

Identity `run` for API symmetry.

Parameters

```
run(m: dictionary) -> dictionary
```

unwrap-or

Extract the inner value or return a default on error.

Parameters

```
unwrap-or(
  default: any,
  m: dictionary
) -> any
```

monad `dictionary`

Result monad instance. `bind` short-circuits on `err`, threading the error value through unchanged.

```
#bind(result.monad, result.ok(3), x =>
result.ok(x * 2))
#bind(result.monad, result.err("oops"), x =>
result.ok(x * 2))
```

```
(tag: "ok", value: 6) (tag: "err",
value: "oops")
```

State instance

`M a = state -> (state, a)`. Threads a mutable-feeling state through a pure computation. This is the engine behind every `free.make`-derived builder.

state

eval

Run a State action and return only the produced value.

Parameters

```
eval(
  m: function,
  init: dictionary
) -> any
```

exec

Run a State action and return only the final state.

Parameters

```
exec(
  m: function,
  init: dictionary
) -> dictionary
```

get-at

Read a single key from a dict-shaped state.

```
#state.run(state.get-at("x", default: 0), (x:
9))
```

```
((x: 9), 9)
```

Parameters

```
get-at(
  key: any,
  default: any
) -> function
```

default `any`

Fallback when key is missing.

Default: none

gets

Read a projected view of the state.

Parameters

```
gets(f: function) -> function
```

f `function`

state -> a.

modify

Apply a transformation to the state.

```
#state.run(state.modify(s => s + (touched:
true)), (a: 1))
```

```
((a: 1, touched: true), none)
```

Parameters

```
modify(f: function) -> function
```

f `function`

state -> state.

modify-at

Apply `f` to a single key of the dict-shaped state. Returns the new value.

```
#state.run(
  state.modify-at("count", x => x + 1,
default: 0),
  (count: 4),
)
```

```
((count: 5), 5)
```

Parameters

```
modify-at(
  key: any,
  f: function,
  default: any
) -> function
```

f `function`

value -> value.

put

Action that overwrites the state, discarding the previous value.

Parameters

```
put(new: dictionary) -> function
```

new `dictionary`

New state.

put-at

Write a single key in a dict-shaped state. Returns the written value.

```
#state.run(state.put-at("x", 10), (:))
```

```
((x: 10), 10)
```

Parameters

```
put-at(
  key: any,
  value: any
) -> function
```

run

Run a State action against an initial state and return `(state, value)`.

```
#state.run(state.put-at("x", 1), (:))
```

```
((x: 1), 1)
```

Parameters

```
run(
  m: function,
  init: dictionary
) -> array
```

monad `dictionary`

State monad instance. Values are functions `state -> (state, a)`. Threads the state through `bind` while producing values per step.

```
#let prog = bind(state.monad, state.put-
at("x", 5), _ =>
  state.get-at("x"))
#state.run(prog, (:))
```

```
((x: 5), 5)
```

get `function`

Action that returns the current state without modifying it.

Reader instance

`M a = env -> a`. Threads an immutable environment — useful for dependency injection without explicit parameter passing.

reader

ask-at

Read a single key from a dict-shaped environment.

Parameters

```
ask-at(
  key: any,
  default: any
) -> function
```

asks

Read a projected view of the environment.

Parameters

```
asks(f: function) -> function
```

f `function`

`env -> a`.

local

Run an action under a temporarily-modified environment.

```
#reader.run(reader.local(e => e + (port: 443),
reader.ask), (host: "x"))
```

```
(host: "x", port: 443)
```

Parameters

```
local(
  f: function,
  m: function
) -> function
```

f `function`

`env -> env`.

m `function`

Action to run under the transformed env.

run

Run a Reader action against an environment.

Parameters

```
run(
  m: function,
  env: dictionary
) -> any
```

monad `dictionary`

Reader monad instance. Values are functions `env -> a`. The environment is read-only — useful for dependency injection without explicit param threading.

```
#let prog = bind(reader.monad, reader.ask, env
=> pure(reader.monad, env.host))
#reader.run(prog, (host: "example.com", port:
80))
```

```
example.com
```

ask `function`

Action that returns the current environment unchanged.

Writer instance

`M a = (log, a)`. Accumulates a log alongside the computed value. The log type must form a monoid; the default instance uses array concatenation.

writer

censor

Apply a transformation to the accumulated log post-hoc, without touching the value.

Parameters

```
censor(
  f: function,
  m: array
) -> array
```

f `function`

`log -> log`.

listen

Run an action and additionally expose its produced log as part of the value: result becomes `(log, (value, log))`.

Parameters

```
listen(m: array) -> array
```

make

Construct a Writer monad for a custom log monoid. Provide an `empty` value and a binary associative `append`. Returns a bundle with the monad instance plus a tailored `tell`.

```
#let w = writer.make(empty: "", append: (a, b)
=> a + b)
#let tell = w.tell
#bind(w.monad, tell("hello "), _ =>
bind(w.monad, tell("world"), _ =>
pure(w.monad, 42)))
```

```
("hello world", 42)
```

Parameters

```
make(
  empty: any,
  append: function
) -> dictionary
```

empty `any`

The monoid identity.

Default: `()`

append `function`

Associative combine.

Default: `(a, b) => a + b`

run

Identity `run` — a Writer value is already `(log, value)`.

Parameters

`run(m: array) -> array`

default `dictionary`

Default Writer bundle: log is an array combined by concatenation.

monad `dictionary`

Default Writer monad instance (array log).

tell `array`

Append a value to the default Writer's log.

```
#writer.tell(("event-1",))
```

```
(("event-1",), none)
```