

Monad (functional programming)

20 languages

[Article](#) [Talk](#)

[Read](#) [Edit](#) [View history](#)

From Wikipedia, the free encyclopedia

For the concept in category theory, see [Monad \(category theory\)](#).

In [functional programming](#), **monads** are a way to structure computations as a sequence of steps, where each step produces a value, plus some extra information about the computation, such as a potential failure, non-determinism, or side effect. More formally, a monad is a [type constructor](#) *M* equipped with two operations, `return` : `<A> (a : A) -> M(A)` which lifts a value into the monadic context, and `bind` : `<A,B> (m_a : M(A), f : A -> M(B)) -> M(B)` which chains monadic computations. In simpler terms, monads can be thought of as [interfaces](#) implemented on type constructors, that allow for functions to abstract over various type constructor variants that implement monad (e.g. `Option`, `List`, etc.).^{[1][2]}

Both the concept of a monad and the term originally come from [category theory](#), where a monad is defined as an [endofunctor](#) with additional structure.^{[a][b]} Research beginning in the late 1980s and early 1990s established that monads could bring seemingly disparate computer-science problems under a unified, functional model. Category theory also provides a few formal requirements, known as the **monad laws**, which should be satisfied by any monad and can be used to [verify](#) monadic code.^{[3][4]}

Since monads make [semantics](#) explicit for a kind of computation, they can also be used to implement convenient language features. Some languages, such as [Haskell](#), even offer pre-built definitions in their core [libraries](#) for the general monad structure and common instances.^{[1][5]}

Overview [\[edit \]](#)

"For a monad `m`, a value of type `m a` represents having access to a value of type `a` within the context of the monad." —C. A. McCann^[6]

More exactly, a monad can be used where unrestricted access to a value is inappropriate for reasons specific to the scenario. In the case of the Maybe monad, it is because the value may not exist. In the case of the [input/output](#) (IO) monad, it is because the value may not be known yet, such as when the monad represents user input that will only be provided after a prompt is displayed. In all cases the scenarios in which access makes sense are captured by the bind operation defined for the monad; for the Maybe monad a value is bound only if it exists, and for the IO monad a value is bound only after the prior operations in the sequence are performed.

A monad can be created by defining a [type constructor](#) *M* and two operations:

- `return :: a -> M a` (often also called *unit*), which receives a value of type `a` and wraps it into a *monadic value* of type `M a`, and
- `bind :: (M a) -> (a -> M b) -> (M b)` (typically represented as `>=>`), which receives a monadic value of type `M a` and a function `f` that accepts values of the base type `a`. Bind unwraps `M a`, applies `f` to it, and can process the result of `f` as a monadic value `M b`.

(An alternative but [equivalent construct](#) using the `join` function instead of the `bind` operator can be found in the later section [§ Derivation from functors](#).)

With these elements, the programmer composes a sequence of function calls (a "pipeline") with several *bind* operators chained together in an expression. Each function call transforms its input plain-type value, and the bind operator handles the returned monadic value, which is fed into the next step in the sequence.

Typically, the bind operator `>=>` may contain code unique to the monad that performs additional computation steps not available in the function received as a parameter. Between each pair of composed function calls, the bind operator can inject into the monadic value `m a` some additional information that is not accessible within the function `f`, and pass it along down the pipeline. It can also exert finer control of the flow of execution, for example by calling the function only under some conditions, or executing the function calls in a particular order.

An example: Maybe [\[edit \]](#)

Further information: [Option type](#)

See also: [Monad \(category theory\) § The maybe monad](#)

One example of a monad is the `Maybe` type. Undefined null results are one particular pain point that many procedural languages don't provide specific tools for dealing with, requiring use of the [null object pattern](#) or checks to test for invalid values at each operation to handle undefined values. This causes bugs and makes it harder to build robust software that gracefully handles errors. The `Maybe` type forces the programmer to deal with these potentially undefined results by explicitly defining the two states of a result: `Just result`, or `Nothing`. For example, the programmer might be constructing a parser, which is to return an intermediate result, or else signal a condition which the parser has detected, and which the programmer must also handle. With just a little extra functional spice on top, this `Maybe` type transforms into a fully-featured monad.^{[\[c\]](#)}12.3 pages 148–151

In most languages, the Maybe monad is also known as an [option type](#), which is just a type that marks whether or not it contains a value. Typically they are expressed as some kind of [enumerated type](#). In the [Rust](#) programming language it is called `Option<T>` and variants of this type can either be a value of [generic type](#) `T`, or the empty variant: `None`.

```
// The <T> represents a generic type "T"
enum Option<T> {
    Some(T),
```

```
    None,  
}
```

`Option<T>` can also be understood as a "wrapping" type, and this is where its connection to monads comes in. In languages with some form of the Maybe type, there are functions that aid in their use such as composing **monadic functions** with each other and testing if a Maybe contains a value.

In the following hard-coded example, a Maybe type is used as a result of functions that may fail, in this case the type returns nothing if there is a [divide-by-zero](#).

```
fn divide(x: Decimal, y: Decimal) -> Option<Decimal> {  
    if y == 0 {  
        None  
    } else {  
        Some(x / y)  
    }  
}  
  
// divide(1.0, 4.0) -> returns Some(0.25)  
// divide(3.0, 0.0) -> returns None
```

One such way to test whether or not a Maybe contains a value is to use `if` statements.

```
let m_x = divide(3.14, 0.0); // see divide function above  
// The if statement extracts x from m_x if m_x is the Just variant of Maybe  
if let Some(x) = m_x {  
    println!("answer: {}", x);  
} else {  
    println!("division failed, divide by zero error...");  
}
```

Other languages may have [pattern matching](#)

```
let result = divide(3.0, 2.0);  
match result {  
    Some(x) => println!("Answer: {}", x),  
    None => println!("division failed; we'll get 'em next time."),  
}
```

Monads can compose functions that return Maybe, putting them together. A concrete example might have one function take in several Maybe parameters, and return a single Maybe whose value is Nothing when any of the parameters are Nothing, as in the following:

```
fn chainable_division(maybe_x: Option<Decimal>, maybe_y: Option<Decimal>) ->  
Option<Decimal> {
```

```

let x = maybe_x?;
let y = maybe_y?;

// If both inputs are non-None,
// check for division by zero and divide accordingly
// otherwise return None
if y == 0 {
    None
} else {
    Some(x / y)
}
}
chainable_division(chainable_division(Some(2.0), Some(0.0)), Some(1.0)); //
inside chainable_division fails, outside chainable_division returns None

```

Instead of repeating `Some` expressions, we can use something called a *bind* operator. (also known as "map", "flatMap", or "shove"^{[8]:2205s}). This operation takes a monad and a function that returns a monad and runs the function on the inner value of the passed monad, returning the monad from the function.

```

// Rust example using ".map". maybe_x is passed through 2 functions that
// return Decimal and String respectively.
// As with normal function composition the inputs and outputs of functions
// feeding into each other should match wrapped types. (i.e. the add_one
// function should return a Decimal which then can be passed to the
// decimal_to_string function)
let maybe_x: Option<Decimal> = Some(1.0);
let maybe_result = maybe_x.map(add_one).map(decimal_to_string);

```

In Haskell, there is an operator *bind*, or (`>=>`) that allows for this monadic composition in a more elegant form similar to [function composition](#).^{[d]:150–151}

```

halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
-- This code halves x twice. it evaluates to Nothing if x is not a multiple
of 4
halve x >=> halve

```

With `>=>` available, `chainable_division` can be expressed much more succinctly with the help of [anonymous functions](#) (i.e. lambdas). Notice in the expression below how the two nested lambdas each operate on the wrapped value in the passed `Maybe` monad using the bind operator.^{[e]:93}

```

chainable_division(mx,my) = mx >=> ( λx -> my >=> (λy -> Just (x / y))
)

```

What has been shown so far is basically a monad, but to be more concise, the following is a strict list of qualities necessary for a monad as defined by the following section.

Monadic Type

A type (`Maybe`)^{[c]:148–151}

Unit operation

A type converter (`Just(x)`)^{[e]:93}

Bind operation

A combinator for monadic functions (`>>=` or `.flatMap()`)^{[d]:150–151}

These are the 3 things needed to form a monad. Other monads may embody different logical processes, and some may have added properties, but all of them will have these three similar components.^{[1][9]}

Definition [\[edit \]](#)

The more common definition for a monad in functional programming, used in the above example, is actually based on a [Kleisli triple](#) $\langle T, \eta, \mu \rangle$ rather than category theory's standard definition. The two constructs turn out to be mathematically equivalent, however, so either definition will yield a valid monad. Given any well-defined basic types T and U , a monad consists of three parts:

- A [type constructor](#) M that builds up a monadic type $M\ T$ ^[f]
- A [type converter](#), often called **unit** or **return**, that embeds an object x in the monad:

`unit : T → M T`^[g]

- A [combinator](#), typically called **bind** (as in [binding a variable](#)) and represented with an [infix operator](#) `>>=` or a method called **flatMap**, that unwraps a monadic variable, then inserts it into a monadic function/expression, resulting in a new monadic value:

`(>>=) : (M T, T → M U) → M U`^[h] so if `ma : M T` and `f : T → M U`, then `(ma >>= f) : M U`

To fully qualify as a monad though, these three parts must also respect a few laws:

- *unit* is a [left-identity](#) for *bind*:

`unit(x) >>= f ↔ f(x)`

- *unit* is also a right-identity for *bind*:

`ma >>= unit ↔ ma`

- *bind* is essentially [associative](#):^[i]

`ma >>= λx → (f(x) >>= g) ↔ (ma >>= f) >>= g`^[1]

Algebraically, this means any monad both gives rise to a category (called the [Kleisli category](#)) and a [monoid](#) in the category of functors (from values to computations), with monadic composition as a binary operator in the monoid^{[8]:2450s} and *unit* as identity in the monoid.

Usage [\[edit \]](#)

The value of the monad pattern goes beyond merely condensing code and providing a link to mathematical reasoning. Whatever language or default [programming paradigm](#) a developer uses, following the monad

pattern brings many of the benefits of [purely functional programming](#). By [reifying](#) a specific kind of computation, a monad not only [encapsulates](#) the tedious details of that computational pattern, but it does so in a [declarative](#) way, improving the code's clarity. As monadic values explicitly represent not only computed values, but computed *effects*, a monadic expression can be replaced with its value in [referentially transparent positions](#), much like pure expressions can be, allowing for many techniques and optimizations based on [rewriting](#).^[4]

Typically, programmers will use *bind* to chain monadic functions into a sequence, which has led some to describe monads as "programmable semicolons", a reference to how many [imperative](#) languages use semicolons to separate [statements](#).^{[1][5]} However, monads do not actually order computations; even in languages that use them as central features, simpler function composition can arrange steps within a program. A monad's general utility rather lies in simplifying a program's structure and improving [separation of concerns](#) through abstraction.^{[4][11]}

The monad structure can also be seen as a uniquely mathematical and [compile time](#) variation on the [decorator pattern](#). Some monads can pass along extra data that is inaccessible to functions, and some even exert finer control over execution, for example only calling a function under certain conditions. Because they let application programmers implement [domain logic](#) while offloading boilerplate code onto pre-developed modules, monads can even be considered a tool for [aspect-oriented programming](#).^[12]

One other noteworthy use for monads is isolating side-effects, like [input/output](#) or mutable [state](#), in otherwise purely functional code. Even purely functional languages *can* still implement these "impure" computations without monads, via an intricate mix of function composition and [continuation-passing style](#) (CPS) in particular.^[2] With monads though, much of this scaffolding can be abstracted away, essentially by taking each recurring pattern in CPS code and bundling it into a distinct monad.^[4]

If a language does not support monads by default, it is still possible to implement the pattern, often without much difficulty. When translated from category-theory to programming terms, the monad structure is a [generic concept](#) and can be defined directly in any language that supports an equivalent feature for [bounded polymorphism](#). A concept's ability to remain agnostic about operational details while working on underlying types is powerful, but the unique features and stringent behavior of monads set them apart from other concepts.^[13]

Applications [\[edit \]](#)

Discussions of specific monads will typically focus on solving a narrow implementation problem since a given monad represents a specific computational form. In some situations though, an application can even meet its high-level goals by using appropriate monads within its core logic.

Here are just a few applications that have monads at the heart of their designs:

- The [Parsec](#) parser library uses monads to combine simpler [parsing](#) rules into more complex ones, and is particularly useful for smaller [domain-specific languages](#).^[14]

- [xmonad](#) is a [tiling window manager](#) centered on the [zipper data structure](#), which itself can be treated monadically as a specific case of [delimited continuations](#).^[15]
- [LINQ](#) by [Microsoft](#) provides a [query language](#) for the [.NET](#) framework that is heavily influenced by functional programming concepts, including core operators for composing queries monadically.^[16]
- [ZipperFS](#) is a simple, experimental [file system](#) that also uses the zipper structure primarily to implement its features.^[17]
- The [Reactive extensions](#) framework essentially provides a (co)monadic interface to [data streams](#) that realizes the [observer pattern](#).^[18]

History [\[edit \]](#)

The term "monad" in programming dates to the [APL](#) and [J](#) programming languages, which tend toward being purely functional. However, in those languages, "monad" is only shorthand for a function taking one parameter (a function with two parameters being a "dyad", and so on).^[19]

The mathematician [Roger Godement](#) was the first to formulate the concept of a monad (dubbing it a "standard construction") in the late 1950s, though the term "monad" that came to dominate was popularized by category-theorist [Saunders Mac Lane](#).^[citation needed] The form defined above using *bind*, however, was originally described in 1965 by mathematician [Heinrich Kleisli](#) to prove that any monad could be characterized as an [adjunction](#) between two (covariant) functors.^[20]

Starting in the 1980s, a vague notion of the monad pattern began to surface in the computer science community. According to programming language researcher [Philip Wadler](#), computer scientist [John C. Reynolds](#) anticipated several facets of it in the 1970s and early 1980s, when he discussed the value of [continuation-passing style](#), of category theory as a rich source for formal semantics, and of the type distinction between values and computations.^[4] The research language [Opal](#), which was actively designed up until 1990, also effectively based I/O on a monadic type, but the connection was not realized at the time.^[21]

The computer scientist [Eugenio Moggi](#) was the first to explicitly link the monad of category theory to functional programming, in a conference paper in 1989,^[22] followed by a more refined journal submission in 1991. In earlier work, several computer scientists had advanced using category theory to provide semantics for the [lambda calculus](#). Moggi's key insight was that a real-world program is not just a function from values to other values, but rather a transformation that forms *computations* on those values. When formalized in category-theoretic terms, this leads to the conclusion that monads are the structure to represent these computations.^[3]

Several others popularized and built on this idea, including Philip Wadler and [Simon Peyton Jones](#), both of whom were involved in the specification of Haskell. In particular, Haskell used a problematic "lazy stream" model up through v1.2 to reconcile I/O with [lazy evaluation](#), until switching over to a more flexible monadic interface.^[23] The Haskell community would go on to apply monads to many problems in functional programming, and in the 2010s, researchers working with Haskell eventually recognized that monads are [applicative functors](#),^{[24][]} and that both monads and [arrows](#) are [monoids](#).^[26]

Analysis [\[edit \]](#)

Verifying the monad laws [\[edit \]](#)

This can be rectified by plugging the specifics of **Maybe** into one side of the general laws, then algebraically building a chain of equalities to reach the other side:

```

Law 3:   (ma >=> f(x)) >=> g(y)                               ⇔   ma >=> (f(x) >=>
g(y))

    if (ma >=> f(x)) is (Just b) then                               if ma is (Just a)
then
    g(ma >=> f(x))
else
    Nothing
end if

    ⇔   if ma is (Just a) and f(a) is (Just b) then
        (g ∘ f) a
    else if ma is (Just a) and f(a) is Nothing then
        Nothing
    else

```

```
Nothing
end if
```

Derivation from functors [\[edit\]](#)

Though rarer in computer science, one can use category theory directly, which defines a monad as a [functor](#) with two added [natural transformations](#).^[k] So to begin, a structure requires a [higher-order function](#) (or "functional") named [map](#) to qualify as a functor:

```
map : (a → b) → (ma → mb)
```

This is not always a major issue, however, especially when a monad is derived from a pre-existing functor, whereupon the monad inherits *map* automatically. (For historical reasons, this `map` is instead called `fmap` in Haskell.)

A monad's first transformation is actually the same *unit* from the Kleisli triple, but following the hierarchy of structures closely, it turns out *unit* characterizes an [applicative functor](#), an intermediate structure between a monad and a basic functor. In the applicative context, *unit* is sometimes referred to as **pure** but is still the same function. What does differ in this construction is the law *unit* must satisfy; as *bind* is not defined, the constraint is given in terms of *map* instead:

```
(unit ∘ φ) x ↔ ((map φ) ∘ unit) x ↔ x[27]
```

The final leap from applicative functor to monad comes with the second transformation, the **join** function (in category theory this is a natural transformation usually called μ), which "flattens" nested applications of the monad:

```
join(mma) : M (M T) → M T
```

As the characteristic function, *join* must also satisfy three variations on the monad laws:

```
(join ∘ (map join)) mma ↔ (join ∘ join) mma ↔ ma
```

```
(join ∘ (map unit)) ma ↔ (join ∘ unit) ma ↔ ma
```

```
(join ∘ (map map φ)) mma ↔ ((map φ) ∘ join) mma ↔ mb
```

Regardless of whether a developer defines a direct monad or a Kleisli triple, the underlying structure will be the same, and the forms can be derived from each other easily:

```
(map φ) ma ↔ ma >>= (unit ∘ φ)
```

```
join(mma) ↔ mma >>= id
```

```
ma >>= f ↔ (join ∘ (map f)) ma[28]
```

Another example: List [\[edit\]](#)

See also: [Monad \(category theory\) § The list and set monads](#)

The **List monad** naturally demonstrates how deriving a monad from a simpler functor can come in handy. In many languages, a list structure comes pre-defined along with some basic features, so a `List` type

constructor and [append](#) operator (represented with `++` for infix notation) are assumed as already given here.

Embedding a plain value in a list is also trivial in most languages:

```
unit(x)  =  [x]
```

From here, applying a function iteratively with a [list comprehension](#) may seem like an easy choice for *bind* and converting lists to a full monad. The difficulty with this approach is that *bind* expects monadic functions, which in this case will output lists themselves; as more functions are applied, layers of nested lists will accumulate, requiring more than a basic comprehension.

However, a procedure to apply any *simple* function over the whole list, in other words *map*, is straightforward:

```
(map φ) xlist  =  [ φ(x1), φ(x2), ..., φ(xn) ]
```

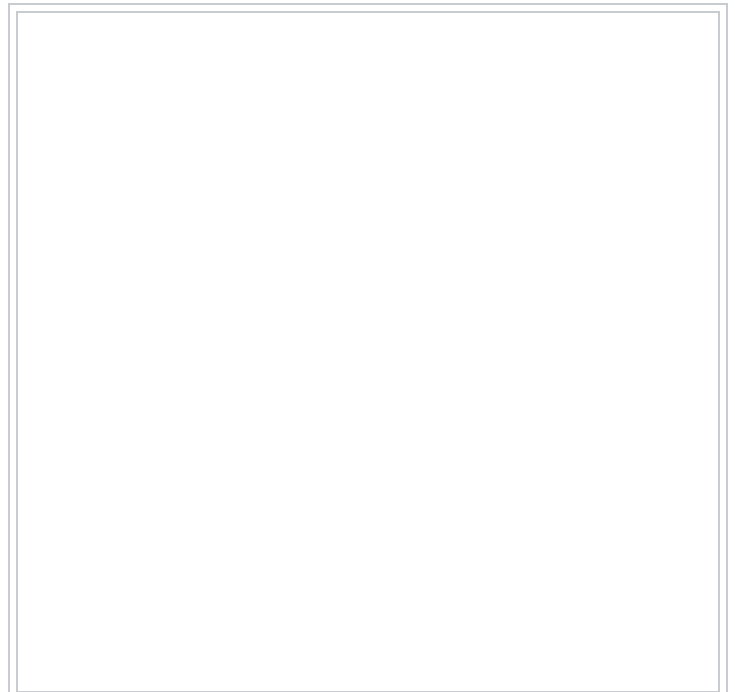
Now, these two procedures already promote `List` to an applicative functor. To fully qualify as a monad, only a correct notion of *join* to flatten repeated structure is needed, but for lists, that just means unwrapping an outer list to append the inner ones that contain values:

```
join(xlistlist)  =  join([xlist1, xlist2, ..., xlistn])  
                  =  xlist1 ++ xlist2 ++ ... ++ xlistn
```

The resulting monad is not only a list, but one that automatically resizes and condenses itself as functions are applied. *bind* can now also be derived with just a formula, then used to feed `List` values through a pipeline of monadic functions:

```
(xlist >>= f)  =  join ◦ (map f)  
xlist
```

One application for this monadic list is representing [nondeterministic computation](#). `List` can hold results for all execution paths in an algorithm, then condense itself at each step to "forget" which paths led to which results (a sometimes important distinction from deterministic, exhaustive algorithms).^{[citation needed](#)}
Another benefit is that checks can be embedded in the monad; specific paths can be pruned transparently at their first point of failure, with no need to rewrite functions in the pipeline.^{[\[28\]](#)}



A second situation where `List` shines is composing [multivalued functions](#). For instance, the n th [complex root](#) of a number should yield n distinct complex numbers, but if another m th root is then taken of those results, the final $m \cdot n$ values should be identical to the output of the $m \cdot n$ th root. `List` completely automates this issue away, condensing the results from each step into a flat, mathematically correct list.^[29]

The `List` monad can greatly simplify the use of multivalued functions, such as complex roots.^[29]

Techniques [\[edit\]](#)

Monads enable various techniques beyond structuring program logic. They provide the basis for specific syntactic features and allow abstraction through their mathematical properties.

Syntactic sugar do-notation [\[edit\]](#)

Although using *bind* openly often makes sense, many programmers prefer a syntax that mimics imperative statements (called *do-notation* in Haskell, *perform-notation* in [OCaml](#), *computation expressions* in [F#](#),^[30] and *for comprehension* in [Scala](#)). This is only [syntactic sugar](#) that disguises a monadic pipeline as a [code block](#); the compiler then automatically translates these expressions into underlying functional code.

Translating the `add` function from the `Maybe` into Haskell can show this feature in action. A non-monadic version of `add` in Haskell looks like this:

```
add mx my =
  case mx of
    Nothing -> Nothing
    Just x   -> case my of
                  Nothing -> Nothing
                  Just y   -> Just (x + y)
```

In monadic Haskell, `return` is the standard name for *unit*, plus lambda expressions must be handled explicitly, but even with these technicalities, the `Maybe` monad makes for a cleaner definition:

```
add mx my =
  mx >>= (\x ->
    my >>= (\y ->
      return (x + y)))
```

With do-notation though, this can be distilled even further into a very intuitive sequence:

```
add mx my = do
  x <- mx
```

```
y <- my
return (x + y)
```

A second example shows how `Maybe` can be used in an entirely different language: F#. With computation expressions, a "safe division" function that returns `None` for an undefined operand *or* division by zero can be written as:

```
let readNum () =
    let s = Console.ReadLine()
    let succ,v = Int32.TryParse(s)
    if (succ) then Some(v) else None

let secure_div =
    maybe {
        let! x = readNum()
        let! y = readNum()
        if (y = 0)
        then None
        else return (x / y)
    }
```

At build-time, the compiler will internally "de-sugar" this function into a denser chain of *bind* calls:

```
maybe.Delay(fun () ->
    maybe.Bind(readNum(), fun x ->
        maybe.Bind(readNum(), fun y ->
            if (y=0) then None else maybe.Return(x / y))))
```

For a last example, even the general monad laws themselves can be expressed in *do*-notation:

```
do { x <- return v; f x } == do { f v }
do { x <- m; return x } == do { m }
do { y <- do { x <- m; f x }; g y } == do { x <- m; y <- f x; g y }
```

General interface [\[edit \]](#)

Every monad needs a specific implementation that meets the monad laws, but other aspects like the relation to other structures or standard idioms within a language are shared by all monads. As a result, a language or library may provide a general `Monad` interface with [function prototypes](#), subtyping relationships, and other general facts. Besides providing a head-start to development and guaranteeing a new monad inherits features from a supertype (such as functors), checking a monad's design against the interface adds another layer of quality control.^{[\[citation needed\]](#)}

Operators [\[edit \]](#)

Monadic code can often be simplified even further through the judicious use of operators. The *map* functional can be especially helpful since it works on more than just ad-hoc monadic functions; so long as a monadic function should work analogously to a predefined operator, *map* can be used to instantly "lift" the simpler operator into a monadic one.^[1] With this technique, the definition of `add` from the `Maybe` example could be distilled into:

```
add(mx,my) = map (+)
```

The process could be taken even one step further by defining `add` not just for `Maybe`, but for the whole `Monad` interface. By doing this, any new monad that matches the structure interface and implements its own *map* will immediately inherit a lifted version of `add` too. The only change to the function needed is generalizing the type signature:

```
add : (Monad Number, Monad Number) → Monad Number[31]
```

Another monadic operator that is also useful for analysis is monadic composition (represented as infix `>=>` here), which allows chaining monadic functions in a more mathematical style:

```
(f >=> g)(x) = f(x) >=> g
```

With this operator, the monad laws can be written in terms of functions alone, highlighting the correspondence to associativity and existence of an identity:

```
(unit >=> g)      ↔ g
(f >=> unit)      ↔ f
(f >=> g) >=> h    ↔ f >=> (g >=> h)[1]
```

In turn, the above shows the meaning of the "do" block in Haskell:

```
do
  _p <- f(x)
  _q <- g(_p)
  h(_q)      ↔ ( f >=> g >=> h )(x)
```

More examples [\[edit \]](#)

Identity monad [\[edit \]](#)

The simplest monad is the **Identity monad**, which just annotates plain values and functions to satisfy the monad laws:

```
newtype Id T = T
```

```
unit(x)      = x  
(x >>= f)    = f(x)
```

`Identity` does actually have valid uses though, such as providing a [base case](#) for recursive [monad transformers](#). It can also be used to perform basic variable assignment within an imperative-style block.^[m]
[\[citation needed\]](#)

Collections [\[edit \]](#)

Any collection with a proper *append* is already a monoid, but it turns out that `List` is not the only [collection](#) that also has a well-defined *join* and qualifies as a monad. One can even mutate `List` into these other monadic collections by simply imposing special properties on *append*:^{[n][o]}

Collection	Monoid properties			Combinatoric properties	
	Commutative?	Idempotent?	Details	Ordered?	Unique items?
List	No	No	Free monoid	Yes	No
Finite multiset	Yes	No		No	No
Finite set	Yes	Yes		No	Yes

IO monad (Haskell) [\[edit \]](#)

As already mentioned, pure code should not have unmanaged side effects, but that does not preclude a program from *explicitly* describing and managing effects. This idea is central to Haskell's **IO monad**, where an object of type `IO a` can be seen as describing an action to be performed in the world, optionally providing information about the world of type `a`. An action that provides no information about the world has the type `IO ()`, "providing" the dummy value `()`. When a programmer binds an `IO` value to a function, the function computes the next action to be performed based on the information about the world provided by the prior action (input from users, files, etc.).^[23] Most significantly, because the value of the IO monad can only be bound to a function that computes another IO monad, the bind function imposes a discipline of a sequence of actions where the result of an action can only be provided to a function that will compute the next action to perform. This means that actions which do not need to be performed never are, and actions that do need to be performed have a well defined sequence.

For example, Haskell has several functions for acting on the wider [file system](#), including one that checks whether a file exists and another that deletes a file. Their two type signatures are:

```
doesFileExist :: FilePath -> IO Bool  
removeFile   :: FilePath -> IO ()
```

The first is interested in whether a given file really exists, and as a result, outputs a [Boolean value](#) within the `IO` monad. The second function, in contrast, is only concerned with acting on the file system so the `IO` container it outputs is empty.

`IO` is not limited just to file I/O though; it even allows for user I/O, and along with imperative syntax sugar, can mimic a typical "Hello, world!" program:

```
main :: IO ()
main = do
  putStrLn "Hello, world!"
  putStrLn "What is your name, user?"
  name <- getLine
  putStrLn ("Nice to meet you, " ++ name ++ "!")
```

Desugared, this translates into the following monadic pipeline (`>>` in Haskell is just a variant of *bind* for when only monadic effects matter and the underlying result can be discarded):

```
main :: IO ()
main =
  putStrLn "Hello, world!" >>
  putStrLn "What is your name, user?" >>
  getLine >>= (\name ->
    putStrLn ("Nice to meet you, " ++ name ++ "!"))
```

Writer monad (Java) [\[edit\]](#)

Another common situation is keeping a [log file](#) or otherwise reporting a program's progress. Sometimes, a programmer may want to log even more specific, technical data for later [profiling](#) or [debugging](#). The **Writer monad** can handle these tasks by generating auxiliary output that accumulates step-by-step.

To show how the monad pattern is not restricted to primarily functional languages, this example implements a `Writer` monad in [Java](#), stored in a class representing the `Writer` monad.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

record Writer<T>(T value, List<String> log) {
  // Internals here...
}
```

Defining *unit* is also very simple:

```
record Writer<T>(T value, List<String> log) {
  // ...
```

```

    public static <T> Writer<T> unit(T value) {
        return new Writer<>(value, new ArrayList<>());
    }
}

```

Only *unit* is needed to define simple functions that output `Writer` objects with debugging notes:

```

import java.util.List;

class Ops {
    private Ops() {}

    public static Writer<Integer> squared(int x) {
        return new Writer<>(x * x, List.of(String.format("%d was squared.",
x)));
    }

    public static Writer<Integer> halved(int x) {
        return new Writer<>(x / 2, List.of(String.format("%d was halved.",
x)));
    }
}

```

A true monad still requires *bind*, but for `Writer`, this operation combines the current log with the log produced by applying a transformation:

```

record Writer<T>(T value, List<String> log) {
    // ...

    public <U> Writer<U> bind(Function<T, Writer<U>> transform) {
        Writer<U> result = transform.apply(this.value);
        List<String> newLog = new ArrayList<>(this.log);
        newLog.addAll(result.log());
        return new Writer<>(result.value(), newLog);
    }
}

```

The sample functions can now be chained together using *bind*, which can be represented using [method chaining](#):

```

Writer<Integer> result = Writer.unit(4)
    .bind(Ops::squared)
    .bind(Ops::halved);

```

The final result is a clean separation of concerns between performing computations and accumulating log output for later inspection:

```
System.out.println(result.value()); // 8
System.out.println(result.log()); // [4 was squared., 16 was halved.]
```

Environment monad [\[edit\]](#)

See also: [Monad \(category theory\) § The environment monad](#)

An environment monad (also called a *reader monad* and a *function monad*) allows a computation to depend on values from a shared environment. The monad type constructor maps a type T to functions of type $E \rightarrow T$, where E is the type of the shared environment. The monad functions are:

return: $T \rightarrow E \rightarrow T = t \mapsto e \mapsto t$

bind: $(E \rightarrow T) \rightarrow (T \rightarrow E \rightarrow T') \rightarrow E \rightarrow T' = r \mapsto f \mapsto e \mapsto f(re)e$

The following monadic operations are useful:

ask: $E \rightarrow E = \text{id}_E$

local: $(E \rightarrow E) \rightarrow (E \rightarrow T) \rightarrow E \rightarrow T = f \mapsto c \mapsto e \mapsto c(fe)$

The ask operation is used to retrieve the current context, while local executes a computation in a modified subcontext. As in a state monad, computations in the environment monad may be invoked by simply providing an environment value and applying it to an instance of the monad.

Formally, a value in an environment monad is equivalent to a function with an added, anonymous argument; return and bind are equivalent to the K and S combinators, respectively, in the [SKI combinator calculus](#).

State monads [\[edit\]](#)

See also: [Monad \(category theory\) § The state monad](#)

A state monad allows a programmer to attach state information of any type to a calculation. Given any value type, the corresponding type in the state monad is a function which accepts a state, then outputs a new state (of type `s`) along with a return value (of type `t`). This is similar to an environment monad, except that it also returns a new state, and thus allows modeling a *mutable* environment.

```
type State s t = s -> (t, s)
```

Note that this monad takes a type parameter, the type of the state information. The monad operations are defined as follows:

```
-- "return" produces the given value without changing the state.
return x = \s -> (x, s)
-- "bind" modifies m so that it applies f to its result.
m >>= f = \r -> let (x, s) = m r in (f x) s
```

Useful state operations include:

```
get = \s -> (s, s) -- Examine the state at this point in the computation.
put s = \_ -> ((), s) -- Replace the state.
modify f = \s -> ((), f s) -- Update the state
```

Another operation applies a state monad to a given initial state:

```
runState :: State s a -> s -> (a, s)
runState t s = t s
```

do-blocks in a state monad are sequences of operations that can examine and update the state data.

Informally, a state monad of state type S maps the type of return values T into functions of type $S \rightarrow T \times S$, where S is the underlying state. The return and bind function are:

return: $T \rightarrow S \rightarrow T \times S = t \mapsto s \mapsto (t, s)$
bind: $(S \rightarrow T \times S) \rightarrow (T \rightarrow S \rightarrow T' \times S) \rightarrow S \rightarrow T' \times S = m \mapsto k \mapsto s \mapsto (k t s')$

From the category theory point of view, a state monad is derived from the adjunction between the product functor and the exponential functor, which exists in any [cartesian closed category](#) by definition.

Continuation monad [\[edit\]](#)

A [continuation monad](#)^[p] with return type R maps type T into functions of type $(T \rightarrow R) \rightarrow R$. It is used to model [continuation-passing style](#). The return and bind functions are as follows:

return: $T \rightarrow (T \rightarrow R) \rightarrow R = t \mapsto f \mapsto f t$
bind: $((T \rightarrow R) \rightarrow R) \rightarrow (T \rightarrow (T' \rightarrow R) \rightarrow R) \rightarrow (T' \rightarrow R) \rightarrow R = c \mapsto f \mapsto k \mapsto c (f t)$

The [call-with-current-continuation](#) function is defined as follows:

call/cc: $((T \rightarrow (T' \rightarrow R) \rightarrow R) \rightarrow (T \rightarrow R) \rightarrow R) \rightarrow (T \rightarrow R) \rightarrow R = f \mapsto k \mapsto (f (t \mapsto k t))$

Program logging [\[edit\]](#)

The following code is pseudocode. Suppose we have two functions `foo` and `bar`, with types

```
foo : int -> int
bar : int -> int
```

That is, both functions take in an integer and return another integer. Then we can apply the functions in succession like so:

```
foo (bar x)
```

Where the result is the result of `foo` applied to the result of `bar` applied to `x`.

But suppose we are debugging our program, and we would like to add logging messages to `foo` and `bar`. So we change the types as so:

```
foo : int -> int * string  
bar : int -> int * string
```

So that both functions return a tuple, with the result of the application as the integer, and a logging message with information about the applied function and all the previously applied functions as the string.

Unfortunately, this means we can no longer `compose` `foo` and `bar`, as their input type `int` is not compatible with their output type `int * string`. And although we can again gain composability by modifying the types of each function to be `int * string -> int * string`, this would require us to add boilerplate code to each function to extract the integer from the tuple, which would get tedious as the number of such functions increases.

Instead, let us define a helper function to abstract away this boilerplate for us:

```
bind : int * string -> (int -> int * string) -> int * string
```

`bind` takes in an integer and string tuple, then takes in a function (like `foo`) that maps from an integer to an integer and string tuple. Its output is an integer and string tuple, which is the result of applying the input function to the integer within the input integer and string tuple. In this way, we only need to write boilerplate code to extract the integer from the tuple once, in `bind`.

Now we have regained some composability. For example:

```
bind (bind (x,s) bar) foo
```

Where `(x,s)` is an integer and string tuple.^[q]

To make the benefits even clearer, let us define an infix operator as an alias for `bind`:

```
(>>=) : int * string -> (int -> int * string) -> int * string
```

So that `t >=> f` is the same as `bind t f`.

Then the above example becomes:

```
((x,s) >=> bar) >=> foo
```

Finally, we define a new function to avoid writing `(x, "")` every time we wish to create an empty logging message, where `""` is the empty string.

```
return : int -> int * string
```

Which wraps `x` in the tuple described above.

The result is a pipeline for logging messages:

```
((return x) >=> bar) >=> foo
```

That allows us to more easily log the effects of `bar` and `foo` on `x`.

`int * string` denotes a pseudo-coded **monadic value**.^[q] `bind` and `return` are analogous to the corresponding functions of the same name. `int * string`, `bind`, and `return` form a monad.

Additive monads [\[edit\]](#)

An **additive monad** is a monad endowed with an added closed, associative, binary operator **mplus** and an **identity element** under *mplus*, called **mzero**. The `Maybe` monad can be considered additive, with `Nothing` as *mzero* and a variation on the **OR** operator as *mplus*. `List` is also an additive monad, with the empty list `[]` acting as *mzero* and the concatenation operator `++` as *mplus*.

Intuitively, *mzero* represents a monadic wrapper with no value from an underlying type, but is also considered a "zero" (rather than a "one") since it acts as an **absorber** for *bind*, returning *mzero* whenever bound to a monadic function. This property is two-sided, and *bind* will also return *mzero* when any value is bound to a monadic **zero function**.

In category-theoretic terms, an additive monad qualifies once as a monoid over monadic functions with *bind* (as all monads do), and again over monadic values via *mplus*.^{[32][r]}

Free monads [\[edit\]](#)

Sometimes, the general outline of a monad may be useful, but no simple pattern recommends one monad or another. This is where a **free monad** comes in; as a **free object** in the category of monads, it can represent monadic structure without any specific constraints beyond the monad laws themselves. Just as a **free**

[monoid](#) concatenates elements without evaluation, a free monad allows chaining computations with markers to satisfy the type system, but otherwise imposes no deeper semantics itself.

For example, by working entirely through the `Just` and `Nothing` markers, the `Maybe` monad is a free monad. The `List` monad, in contrast, is not a free monad since it brings extra, specific facts about lists (like *append*) into its definition. One last example is an abstract free monad:

```
data Free f a
  = Pure a
  | Free (f (Free f a))

unit :: a -> Free f a
unit x = Pure x

bind :: Functor f => Free f a -> (a -> Free f b) -> Free f b
bind (Pure x) f = f x
bind (Free x) f = Free (fmap (\y -> bind y f) x)
```

Free monads, however, are *not* restricted to a linked-list like in this example, and can be built around other structures like [trees](#).

Using free monads intentionally may seem impractical at first, but their formal nature is particularly well-suited for syntactic problems. A free monad can be used to track syntax and type while leaving semantics for later, and has found use in parsers and [interpreters](#) as a result.^[33] Others have applied them to more dynamic, operational problems too, such as providing [iteratees](#) within a language.^[34]

Comonads [\[edit\]](#)

Besides generating monads with extra properties, for any given monad, one can also define a **comonad**. Conceptually, if monads represent computations built up from underlying values, then comonads can be seen as reductions back down to values. Monadic code, in a sense, cannot be fully "unpacked"; once a value is wrapped within a monad, it remains quarantined there along with any side-effects (a good thing in purely functional programming). Sometimes though, a problem is more about consuming contextual data, which comonads can model explicitly.

Technically, a comonad is the [categorical dual](#) of a monad, which loosely means that it will have the same required components, only with the direction of the type signatures *reversed*. Starting from the *bind*-centric monad definition, a comonad consists of:

- A type constructor W that marks the higher-order type $W\ T$
- The dual of *unit*, called **counit** here, extracts the underlying value from the comonad:

```
counit(wa) : W T → T
```

- A reversal of *bind* (also represented with `=>>`) that **extends** a chain of reducing functions:

```
(wa ==>> f) : (W U, W U → T) → W T[s]
```

extend and *counit* must also satisfy duals of the monad laws:

```
counit . ( (wa ==>> f) → wb ) ↔ f(wa) → b
wa ==>> counit ↔ wa
wa ( (==>> f(wx = wa)) → wb (==>> g(wy = wb)) → wc ) ↔ ( wa (==>> f(wx = wa))
→ wb ) (==>> g(wy = wb)) → wc
```

Analogous to monads, comonads can also be derived from functors using a dual of *join*:

- **duplicate** takes an already comonadic value and wraps it in another layer of comonadic structure:

```
duplicate(wa) : W T → W (W T)
```

While operations like *extend* are reversed, however, a comonad does *not* reverse functions it acts on, and consequently, comonads are still functors with *map*, not *cofunctors*. The alternate definition with *duplicate*, *counit*, and *map* must also respect its own comonad laws:

```
((map duplicate) . duplicate) wa ↔ (duplicate . duplicate) wa ↔ wwwa
((map counit) . duplicate)      wa ↔ (counit . duplicate)      wa ↔ wa
((map map φ) . duplicate)       wa ↔ (duplicate . (map φ))      wa ↔ wwφ
```

And as with monads, the two forms can be converted automatically:

```
(map φ) wa      ↔ wa ==>> (φ . counit) wx
duplicate wa    ↔ wa ==>> wx
```

```
wa ==>> f(wx) ↔ ((map f) . duplicate) wa
```

A simple example is the **Product comonad**, which outputs values based on an input value and shared environment data. The `Product` comonad is only the dual of the `Writer` monad and effectively the same as the `Reader` monad (both discussed below). `Product` and `Reader` differ only in which function signatures they accept, and how they complement those functions by wrapping or unwrapping values.

A less trivial example is the **Stream comonad**, which can be used to represent [data streams](#) and attach filters to the incoming signals with *extend*. While not as popular as monads, researchers have found comonads particularly useful for [stream processing](#) and modeling [dataflow programming](#).^{[35][36]}

Due to their strict definitions, however, one cannot simply move objects back and forth between monads and comonads. As an even higher abstraction, [arrows](#) can subsume both structures, but finding more granular

ways to combine monadic and comonadic code is an active area of research.^{[37][38]}

See also [\[edit \]](#)

Alternatives for modeling computations:

- [Effect systems](#) (particularly algebraic effect handlers) are a different way to describe side effects as types
- [Uniqueness types](#) are a third approach to handling side-effects in functional languages

Related design concepts:

- [Aspect-oriented programming](#) emphasizes separating out ancillary bookkeeping code to improve modularity and simplicity
- [Inversion of control](#) is the abstract principle of calling specific functions from an overarching framework
- [Type classes](#) are a specific language feature used to implement monads and other structures in Haskell
- The [decorator pattern](#) is a more concrete, ad-hoc way to achieve similar benefits in object-oriented programming

Generalizations of monads:

- [Applicative functors](#) generalize from monads by keeping only *unit* and laws relating it to *map*
- [Arrows](#) use added structure to bring plain functions and monads under a single interface
- [Monad transformers](#) act on distinct monads to combine them modularly

Notes [\[edit \]](#)

- [^] More formally, a monad is a [monoid](#) in the category of [endofunctors](#).
- [^] Because functions on multiple [free variables](#) are common in programming, monads as described in this article are technically what category theorists would call [strong monads](#).^[3]
- [^] ^{[a](#)} ^{[b](#)} Specific motivation for Maybe can be found in (Hutton 2016).^[7]
- [^] ^{[a](#)} ^{[b](#)} Hutton abstracts a `bind` which when given a type *a* that may fail, and a mapping *a* → *b* that may fail, produces a result *b* that may fail. (Hutton, 2016)^[7]
- [^] ^{[a](#)} ^{[b](#)} (Hutton 2016) notes that Just might denote Success, and Nothing might denote Failure.^[7]
- [^] Semantically, *M* is not trivial and represents an [endofunctor](#) over the [category](#) of all well-typed values:
$$M : Val \rightarrow Val$$
- [^] While a (parametrically polymorphic) function in programming terms, *unit* (often called η in category theory) is mathematically a [natural transformation](#), which maps between *functors*: $\eta_A : \text{id}(Val_A) \rightarrow M(Val_A)$
- [^] *bind*, in contrast, is not a natural transformation in category theory, but rather an extension $-^*$ that [lifts](#) a mapping (from values to computations) into a morphism between computations:
$$\forall f : Val_A \rightarrow M(Val_B), f^* : M(Val_A) \rightarrow M(Val_B)$$
- [^] Strictly speaking, *bind* may not be formally associative in all contexts because it corresponds to application within [lambda calculus](#), not mathematics. In rigorous lambda-calculus, evaluating a *bind* may require first wrapping the right term (when binding two monadic values) or the bind itself (between two monadic functions) in an [anonymous function](#) to still accept input from the left.^[10]
- [^] By GHC version 7.10.1, and going forward, Haskell began enforcing Haskell's 2014 Applicative Monad proposal (AMP) which requires the insertion of 7 lines of code into any existing modules which use monads.^[25]

- k. [^] These natural transformations are usually denoted as morphisms η , μ . That is: η , μ denote *unit*, and *join* respectively.
- l. [^] Some languages like Haskell even provide a pseudonym for *map* in other contexts called `lift`, along with multiple versions for different parameter counts, a detail ignored here.
- m. [^] In category theory, the `Identity` monad can also be viewed as emerging from *adjunction* of any functor with its inverse.
- n. [^] Category theory views these collection monads as adjunctions between the *free functor* and different functors from the *category of sets* to the *category of monoids*.
- o. [^] Here the task for the programmer is to construct an appropriate monoid, or perhaps to choose a monoid from a library.
- p. [^] The reader may wish to follow McCann's thread^[6] and compare it with the Types below.
- q. [^] ^{a b} In this case, the `bind` has *pasted* in a `string` where previously only an `integer` had been; that is, the programmer has constructed an *adjunction*: a tuple `(x,s)`, denoted `int * string` in the pseudocode § above.
- r. [^] Algebraically, the relationship between the two (non-commutative) monoid aspects resembles that of a *near-semiring*, and some additive monads do qualify as such. However, not all additive monads meet the *distributive* laws of even a near-semiring.^[32]
- s. [^] In Haskell, *extend* is actually defined with the inputs swapped, but as currying is not used in this article, it is defined here as the exact dual of *bind*.

References [edit]

-
1. [^] ^{a b c d e f} O'Sullivan, Bryan; Goerzen, John; Stewart, Don (2009). "Monads" . *Real World Haskell* . Sebastopol, California: O'Reilly Media. chapter 14. ISBN 978-0-596-51498-3.
 2. [^] ^{a b} Wadler, Philip (June 1990). *Comprehending Monads*. ACM Conference on LISP and Functional Programming. Nice, France. CiteSeerX 10.1.1.33.5381 . {{cite conference}}: Cite uses deprecated parameter |citeseerx= (help)
 3. [^] ^{a b c} Moggi, Eugenio (1991). "Notions of computation and monads" (PDF). *Information and Computation*. **93** (1): 55–92. doi:10.1016/0890-5401(91)90052-4 .
 4. [^] ^{a b c d e} Wadler, Philip (January 1992). *The essence of functional programming*. 19th Annual ACM Symposium on Principles of Programming Languages. Albuquerque, New Mexico. CiteSeerX 10.1.1.38.9516 . {{cite conference}}: Cite uses deprecated parameter |citeseerx= (help)
 5. [^] ^{a b} Hudak, Paul; Peterson, John; Fasel, Joseph (1999). "About Monads" . *A Gentle Introduction to Haskell* 98 . chapter 9.
 6. [^] ^{a b} C. A. McCann's answer (Jul 23 '10 at 23:39) How and why does the Haskell Cont monad work?
 7. [^] ^{a b c} Graham Hutton (2016) *Programming in Haskell* 2nd Edition
 8. [^] ^{a b} Beckman, Brian (21 November 2012). "Don't fear the Monad" . *YouTube*.
 9. [^] Spivey, Mike (1990). "A functional theory of exceptions" (PDF). *Science of Computer Programming*. **14** (1): 25–42. doi:10.1016/0167-6423(90)90056-J .
 10. [^] "Monad laws" . *HaskellWiki*. haskell.org. Retrieved 14 October 2018.
 11. [^] "What a Monad is not" . 7 October 2018.
 12. [^] De Meuter, Wolfgang (1997). *Monads as a theoretical foundation for AOP* (PDF). International Workshop on Aspect Oriented Programming at ECOOP. Jyväskylä, Finland.
 13. [^] "Monad (sans metaphors)" . *HaskellWiki*. 1 November 2009. Retrieved 24 October 2018.

14. [^] O'Sullivan, Bryan; Goerzen, John; Stewart, Don (2009). "Using Parsec" . *Real World Haskell* . Sebastopol, California: O'Reilly Media. chapter 16. ISBN 978-0-596-51498-3.
15. [^] Stewart, Don (17 May 2007). "Roll Your Own Window Manager: Tracking Focus with a Zipper" . *Control.Monad.Writer*. Archived from the original on 20 February 2018. Retrieved 19 November 2018.
16. [^] Benton, Nick (2015). "Categorical Monads and Computer Programming" (PDF). *London Mathematical Society Impact150 Stories*. **1**. Retrieved 19 November 2018.
17. [^] Kiselyov, Olag (2007). "Delimited Continuations in Operating Systems". *Modeling and Using Context*. Lecture Notes in Computer Science. Vol. 4635. Springer Berlin Heidelberg. pp. 291–302. doi:10.1007/978-3-540-74255-5_22 . ISBN 978-3-540-74255-5.
18. [^] Meijer, Erik (27 March 2012). "Your Mouse is a Database" . *ACM Queue*. **10** (3): 20–33. doi:10.1145/2168796.2169076 .
19. [^] Iverson, Kenneth (September 1987). "A dictionary of APL" . *APL Quote Quad*. **18** (1): 5–40. doi:10.1145/36983.36984 . ISSN 1088-6826 . S2CID 18301178 . Retrieved 19 November 2018.
20. [^] Kleisli, Heinrich (1965). "Every standard construction is induced by a pair of adjoint functors" (PDF). *Proceedings of the American Mathematical Society*. **16** (3): 544–546. doi:10.1090/S0002-9939-1965-0177024-4 . Retrieved 19 November 2018.
21. [^] Pepper, Peter, ed. (November 1997). *The Programming Language Opal* (Technical report) (5th corrected ed.). Fachbereich Informatik, Technische Universität Berlin. CiteSeerX 10.1.1.40.2748 . {{cite tech report}}: Cite uses deprecated parameter |citeseerx= (help)
22. [^] Moggi, Eugenio (June 1989). *Computational lambda-calculus and monads* (PDF). Fourth Annual Symposium on Logic in computer science. Pacific Grove, California.
23. [^] ^a ^b Peyton Jones, Simon L.; Wadler, Philip (January 1993). *Imperative functional programming* (PDF). 20th Annual ACM Symposium on Principles of Programming Languages. Charleston, South Carolina.
24. [^] Brent Yorgey [Typeclassopedia](#)
25. [^] Stack overflow (8 Sep 2017) [Defining a new monad in haskell raises no instance for Applicative](#)
26. [^] Brent Yorgey [Monoids](#)
27. [^] "Applicative functor" . *HaskellWiki*. Haskell.org. 7 May 2018. Archived from the original on 30 October 2018. Retrieved 20 November 2018.
28. [^] ^a ^b Gibbard, Cale (30 December 2011). "Monads as containers" . *HaskellWiki*. Haskell.org. Archived from the original on 14 December 2017. Retrieved 20 November 2018.
29. [^] ^a ^b Piponi, Dan (7 August 2006). "You Could Have Invented Monads! (And Maybe You Already Have.)" . *A Neighborhood of Infinity*. Archived from the original on 24 October 2018. Retrieved 16 October 2018.
30. [^] "Some Details on F# Computation Expressions" . 21 September 2007. Retrieved 9 October 2018.
31. [^] Giles, Brett (12 August 2013). "Lifting" . *HaskellWiki*. Haskell.org. Archived from the original on 29 January 2018. Retrieved 25 November 2018.
32. [^] ^a ^b Rivas, Exequiel; Jaskelioff, Mauro; Schrijvers, Tom (July 2015). *From monoids to near-semirings: the essence of MonadPlus and Alternative* (PDF). 17th International ACM Symposium on Principles and Practice of Declarative Programming. Siena, Italy.
33. [^] Swierstra, Wouter (2008). "Data types à la carte" (PDF). *Functional Pearl. Journal of Functional Programming*. **18** (4). Cambridge University Press: 423–436. doi:10.1017/s0956796808006758 . ISSN 1469-7653 . S2CID 21038598 .
34. [^] Kiselyov, Oleg (May 2012). Schrijvers, Tom; Thiemann, Peter (eds.). *Iteratees* (PDF). International Symposium on Functional and Logic Programming. Lecture Notes in Computer Science. Vol. 7294. Kobe, Japan: Springer-Verlag. pp. 166–181. doi:10.1007/978-3-642-29822-6_15 . ISBN 978-3-642-29822-6.

35. [^] Uustalu, Tarmo; Vene, Varmo (July 2005). Horváth, Zoltán (ed.). *The Essence of Dataflow Programming* (PDF). First Summer School, Central European Functional Programming. Lecture Notes in Computer Science. Vol. 4164. Budapest, Hungary: Springer-Verlag. pp. 135–167. ISBN 978-3-540-46845-5.

36. [^] Uustalu, Tarmo; Vene, Varmo (June 2008). "Comonadic Notions of Computation" . *Electronic Notes in Theoretical Computer Science*. **203** (5). Elsevier: 263–284. doi:10.1016/j.entcs.2008.05.029 . ISSN 1571-0661 .

37. [^] Power, John; Watanabe, Hiroshi (May 2002). "Combining a monad and a comonad" (PDF). *Theoretical Computer Science*. **280** (1–2). Elsevier: 137–162. doi:10.1016/s0304-3975(01)00024-x . ISSN 0304-3975 .

38. [^] Gaboardi, Marco; Katsumata, Shin-ya; Orchard, Dominic; Breuvert, Flavien; Uustalu, Tarmo (September 2016). *Combining Effects and Coeffects via Grading* (PDF). 21st ACM International Conference on Functional Programming. Nara, Japan: Association for Computing Machinery. pp. 476–489. doi:10.1145/2951913.2951939 . ISBN 978-1-4503-4219-3.

External links [edit]

HaskellWiki references:

- "[All About Monads](#) " (originally by Jeff Newbern) — A comprehensive discussion of all the common monads and how they work in Haskell; includes the "mechanized assembly line" analogy.
- "[Typeclassopedia](#) " (originally by Brent Yorgey) — A detailed exposition of how the leading typeclasses in Haskell, including monads, interrelate.

The Wikibook *Haskell* has a page on the topic of: *[Understanding monads](#)*

Tutorials:

- "[A Fistful of Monads](#) " (from the online Haskell textbook *Learn You a Haskell for Great Good!* — A chapter introducing monads from the starting-point of functor and applicative functor typeclasses, including examples.
- "[For a Few Monads More](#) " — A second chapter explaining more details and examples, including a Probability monad for [Markov chains](#).
- "[Functors, Applicatives, And Monads In Pictures](#) " (by Aditya Bhargava) — A quick, humorous, and visual tutorial on monads.

Interesting cases:

- "[UNIX pipes as IO monads](#) " (by Oleg Kiselyov) — A short essay explaining how [Unix pipes](#) are effectively monadic.
- [Pro Scala: Monadic Design Patterns for the Web](#)* (by Gregory Meredith) — An unpublished, full-length manuscript on how to improve many facets of web development in [Scala](#) with monads.

V · T · E	Software design patterns	[show]
V · T · E	Formal semantics (natural language)	[show]
V · T · E	Programming paradigms	[show]

This page was last edited on 13 September 2026, at 10:59 (UTC). Page was rendered with [Parsoid](#).

Text is available under the [Creative Commons Attribution-ShareAlike 4.0 License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.