

Haskell CheatSheet

Hello, Home!

```
main = do putStr "What's your name? "
          name <- getLine
          putStrLn ("It's 2020, " ++ name ++ "! Stay home, stay safe!")
```

Pattern Matching

Functions can be defined using the usual `if_then_else_` construct, or as expressions *guarded* by Boolean expressions as in mathematics, or by *pattern matching* —a form of ‘syntactic comparison’.

```
fact n = if n == 0 then 1 else n * fact (n - 1)
```

```
fact' n | n == 0 = 1
        | n != 0 = n * fact' (n - 1)
```

```
fact'' 0 = 1
fact'' n = n * fact'' (n - 1)
```

The above definitions of the factorial function are all equal.

Guards, as in the second version, are a form of ‘multi-branching conditional’.

In the final version, when a call, say, `fact 5` happens we compare *syntactically* whether 5 and the first pattern 0 are the same. They are not, so we consider the second case with the understanding that an identifier appearing in a pattern matches *any* argument, so the second clause is used.

Hence, when pattern matching is used, order of equations matters: If we declared the `n`-pattern first, then the call `fact 0` would match it and we end up with `0 * fact (-1)`, which is not what we want!

If we simply defined the final `fact` using *only* the first clause, then `fact 1` would crash with the error *Non-exhaustive patterns in function fact*. That is, we may define *partial functions* by not considering all possible shapes of inputs.

See also “[view patterns](#)”.

Local Bindings

An equation can be qualified by a `where` or `let` clause for defining values or functions used only within an expression.

```
...e...e...e where e = expr
≈ let e = expr in ...expr...expr...expr
```

It sometimes happens in functional programs that one clause of a function needs *part of* an argument, while another operators on the *whole* argument. It is tedious (and inefficient) to write out the structure of the complete argument again when referring to it. Use the “as operator” `@` to label all or part of an argument, as in

```
f label@(x:y:ys) = ...
```

Operators

Infix operators in Haskell must consist entirely of ‘symbols’ such as `&`, `^`, `!`, `...` rather than alphanumeric characters. Hence, while addition, `+`, is written infix, integer division is written prefix with `div`.

We can always use whatever fixity we like:

- ◊ If `f` is any *prefix* binary function, then `x 'f' y` is a valid *infix* call.
- ◊ If `⊕` is any *infix* binary operator, then `(⊕) x y` is a valid *prefix* call.

It is common to fix one argument ahead of time, e.g., $\lambda x \rightarrow x + 1$ is the successor operation and is written more tersely as `(+1)`. More generally, $(\oplus r) = \lambda x \rightarrow x \oplus r$.

The usual arithmetic operations are `+`, `/`, `*`, `-` but `%` is used to make fractions.

The Boolean operations are `==`, `/=`, `&&`, `||` for equality, discrepancy, conjunction, and disjunction.

Types

Type are inferred, but it is better to write them explicitly so that *you communicate your intentions to the machine*. If you *think* that expression e has type τ then write $e :: \tau$ to *communicate* that to the machine, which will silently accept your claim or reject it loudly.

Type	Name	Example Value
Small integers	Int	42
Unlimited integers	Integer	7376541234
Reals	Float	3.14 and $2 \% 5$
Booleans	Boolean	True and False
Characters	Char	'a' and '3'
Strings	String	"salam"
Lists	$[\alpha]$	[] or $[x_1, \dots, x_n]$
Tuples	(α, β, γ)	(x_1, x_2, x_3)
Functions	$\alpha \rightarrow \beta$	$\lambda x \rightarrow \dots$

Polymorphism is the concept that allows one function to operate on different types.

- ◊ A function whose type contains *variables* is called a *polymorphic function*.
- ◊ The simplest polymorphic function is $\text{id} :: a \rightarrow a$, defined by $\text{id } x = x$.

Tuples

Tuples $(\alpha_1, \dots, \alpha_n)$ are types with values written (x_1, \dots, x_n) where each $x_i :: \alpha_i$. They are a form of ‘record’ or ‘product’ type.

E.g., $(\text{True}, 3, \text{'a'}) :: (\text{Boolean}, \text{Int}, \text{Char})$.

Tuples are used to “return multiple values” from a function.

Two useful functions on tuples of length 2 are:

```
fst :: ( $\alpha, \beta$ )  $\rightarrow \alpha$ 
fst (x, y) = x
```

```
snd :: ( $\alpha, \beta$ )  $\rightarrow \beta$ 
snd (x, y) = y
```

If in addition you `import Control.Arrow` then you may use:

```
first :: ( $\alpha \rightarrow \tau$ )  $\rightarrow (\alpha, \beta) \rightarrow (\tau, \beta)$ 
first f (x, y) = (f x, y)
```

```
second :: ( $\beta \rightarrow \tau$ )  $\rightarrow (\alpha, \beta) \rightarrow (\alpha, \tau)$ 
second g (x, y) = (x, g y)
```

```
(***) :: ( $\alpha \rightarrow \alpha'$ )  $\rightarrow (\beta \rightarrow \beta) \rightarrow (\alpha, \beta) \rightarrow (\alpha', \beta')$ 
(f *** g) (x, y) = (f x, g y)
```

```
(&&&) :: ( $\tau \rightarrow \alpha$ )  $\rightarrow (\tau \rightarrow \beta) \rightarrow \tau \rightarrow (\alpha, \beta)$ 
(f &&& g) x = (f x, g x)
```

Lists

Lists are sequences of items of the same type. If each $x_i :: \alpha$ then $[x_1, \dots, x_n] :: [\alpha]$.

Lists are useful for functions that want to ‘non-deterministically’ return a value: They return a list of all possible values.

- ◊ The *empty list* is []
- ◊ We “construct” nonempty lists using $(:)$ $:: \alpha \rightarrow [\alpha] \rightarrow [\alpha]$
- ◊ Abbreviation: $[x_1, \dots, x_n] = x_1 : (x_2 : (\dots (x_n : [])))$
- ◊ *List comprehensions*: $[f \ x \mid x \leftarrow xs, p \ x]$ is the list of elements $f \ x$ where x is an element from list xs and x satisfies the property p
 - ◊ E.g., $[2 * x \mid x \leftarrow [2, 3, 4], x < 4] \approx [2 * 2, 2 * 3] \approx [4, 6]$
- ◊ Shorthand notation for segments: u may be omitted to yield *infinite lists*
 - ◊ $[1 .. u] = [1, 1 + 1, 1 + 2, \dots, u]$.
 - ◊ $[a, b, .., u] = [a + i * \text{step} \mid i \leftarrow [0 .. u - a]]$ where $\text{step} = b - a$

Strings are just lists of characters: $"c_0c_1\dots c_n" \approx ['c_0', \dots, 'c_n']$.

- ◊ Hence, all list methods work for strings.

Pattern matching on lists

```
prod [] = 1
prod (x:xs) = x * prod xs

fact n = prod [1 .. n]
```

If your function needs a case with a list of say, length 3, then you can match directly on that *shape* via `[x, y, z]` —which is just an abbreviation for the shape `x:y:z:[]`. Likewise, if we want to consider lists of length *at least* 3 then we match on the shape `x:y:z:zs`. E.g., define the function that produces the maximum of a non-empty list, or the function that removes adjacent duplicates —both require the use of guards.

```
[x0, ..., xn] !! i = xi
[x0, ..., xn] ++ [y0, ..., ym] = [x0, ..., xn, y0, ..., ym]
concat [xs0, ..., xsn] = xs0 ++ ... ++ xsn

{- Partial functions -}
head [x0, ..., xn] = x0
tail [x0, ..., xn] = [x1, ..., xn]
init [x0, ..., xn] = [x0, ..., xn-1]
last [x0, ..., xn] = xn

take k [x0, ..., xn] = [x0, ..., xk-1]
drop k [x0, ..., xn] = [xk, ..., xn]

sum [x0, ..., xn] = x0 + ... + xn
prod [x0, ..., xn] = x0 * ... * xn
reverse [x0, ..., xn] = [xn, ..., x0]
elem x [x0, ..., xn] = x == x0 || ... || x == xn

zip [x0, ..., xn] [y0, ..., ym] = [(x0, y0), ..., (xk, yk)] where k = n 'min' m
unzip [(x0, y0), ..., (xk, yk)] = ([x0, ..., xk], [y0, ..., yk])
```

Duality: Let $\partial f = \text{reverse} \circ f \circ \text{reverse}$, then $\text{init} = \partial \text{tail}$ and $\text{take } k = \partial (\text{drop } k)$; even $\text{pure} \circ \text{head} = \partial (\text{pure} \circ \text{last})$ where $\text{pure } x = [x]$.

List ‘Design Patterns’

Many functions have the same ‘form’ or ‘design pattern’, a fact which is taken advantage of by defining *higher-order functions* to factor out the structural similarity of the individual functions.

```
map f xs = [f x | x <- xs]
```

- ◊ Transform all elements of a list according to the function `f`.

```
filter p xs = [x | x <- xs, p x]
```

- ◊ Keep only the elements of the list that satisfy the predicate `p`.
- ◊ `takeWhile p xs` \approx Take elements of `xs` that satisfy `p`, but stop at the first element that does not satisfy `p`.
- ◊ `dropWhile p xs` \approx Drop all elements until you see one that does not satisfy the predicate.
- ◊ `xs = takeWhile p xs ++ dropWhile p xs`.

Right-folds let us ‘sum’ up the elements of the list, associating to the right.

```
foldr (⊕) e ≈ λ (x0 : (x1 : (... : (xn : []))))
  → (x0 ⊕ (x1 ⊕ (... ⊕ (xn ⊕ e))))
```

This function just replaces cons “`:`” and `[]` with \oplus and `e`. That’s all.

- ◊ E.g., replacing `:`, `[]` with themselves does nothing: `foldr (:) [] = id`.

All functions on lists can be written as folds!

```
h [] = e  &  h (x:xs) = x ⊕ h xs
≡ h = foldr (λ x rec_call → x ⊕ rec_call) e
  ◊ Look at the two cases of a function and move them to the two first arguments of the fold.
    ◊ map f = foldr (λ x ys → f x : ys) []
    ◊ filter p = foldr (λ x ys → if (p x) then (x:ys) else ys) []
    ◊ takeWhile p = foldr (λ x ys → if (p x) then (x:ys) else []) []
```

You can also fold leftward, i.e., by associating to the left:

```
foldl (⊕) e  ≈  λ (x0 : (x1 : (... : (xn : []))))
                → (((e ⊕ x0) ⊕ x1) ⊕ ... ) ⊕ xn
```

Unless the operation \oplus is associative, the folds are generally different.

- ◊ E.g., `foldl (/) 1 [1..n] ≈ 1 / n!` where `n ! = product [1..n]`.
- ◊ E.g., `-55 = foldl (-) 0 [1..10] ≠ foldr (-) 0 [1..10] = -5`.

If h swaps arguments — $h(x \oplus y) = h y \oplus h x$ — then h swaps folds:

$h . foldr (\oplus) e = foldl (\ominus) e'$ where $e' = h e$ and $x \ominus y = x \oplus h y$.

E.g., `foldl (-) 0 xs = - (foldr (+) 0 xs) = - (sum xs)`
 and `n ! = foldr (*) 1 [1..n] = 1 / foldl (/) 1 [1..n]`.

(Floating points are a leaky abstraction!)

Algebraic data types

When we have ‘possible scenarios’, we can make a type to consider each option. E.g., `data Door = Open | Closed` makes a new datatype with two different values. Under the hood, `Door` could be implemented as integers and `Open` is 0 and `Closed` is 1; or any other implementation —*all that matters* is that we have a new type, `Door`, with two different values, `Open` and `Closed`.

Usually, our scenarios contain a ‘payload’ of additional information; e.g., `data Door2 = Open | Ajar Int | Closed`. Here, we have a new way to construct `Door` values, such as `Ajar 10` and `Ajar 30`, that we could interpret as denoting how far the door is open/. Under the hood, `Door2` could be implemented as pairs of integers, with `Open` being (0,0), `Ajar n` being (1, n), and `Closed` being (2, 0) —i.e., as the pairs “(value position, payload data)”. Unlike functions, a value construction such as `Ajar 10` cannot be simplified any further; just as the list value `1:2:3:[]` cannot be simplified any further. Remember, the representation under the hood does not matter, what matters is that we have three possible *construction forms* of `Door2` values.

Languages, such as C, which do not support such an “algebraic” approach, force you, the user, to actually choose a particular representation —even though, it does not matter, since we only want *a way to speak of* “different cases, with additional information”.

In general, we declare the following to get an “enumerated type with payloads”.

```
data D = C0 τ1 τ2 ... τm | C1 ... | Cn ... deriving Show
```

There are n constructors C_i that make *different* values of type D ; e.g., $C_0 x_1 x_2 \dots x_m$ is a D -value whenever each x_i is a τ_i -value. The “`deriving Show`” at the end of the definition is necessary for user-defined types to make sure that values of these types can be printed in a standard form.

We may now define functions on D by pattern matching on the possible ways to *construct* values for it; i.e., by considering the cases C_i .

In-fact, we could have written `data D α1 α2 ... αk = ...`, so that we speak of “ D values *parameterised* by types α_i ”. E.g., “lists whose elements are of type α ” is defined by `data List α = Nil | Cons α (List α)` and, for example, `Cons 1 (Cons 2 Nil)` is a value of `List Int`, whereas `Cons 'a' Nil` is of type `List Char`. —The `List` type is missing the “`deriving Show`”, see below for how to *mixin* such a feature.

For example, suppose we want to distinguish whether we have an α -value or a β -value, we use `Either`. Let’s then define an example *infix* function using pattern matching.

```
data Either α β = Left α | Right β

(+++) :: (α → α') → (β → β') → Either α β → Either α' β'
(f +++ g) (Left x) = Left $ f x
(f +++ g) (Right x) = Right $ g x

right :: (β → τ) → Either α β → Either α τ
right f = id +++ f
```

The above `(+++)` can be found in `Control.Arrow` and is also known as `either` in the standard library.

Typeclasses and overloading

Overloading is using the same name to designate operations “of the same nature” on values of different types.

E.g., the `show` function converts its argument into a string; however, it is not polymorphic: We cannot define `show :: $\alpha \rightarrow \text{String}$` with one definition since some items, like functions or infinite datatypes, cannot be printed and so this is not a valid type for the function `show`.

Haskell solves this by having `Show` typeclass whose *instance types* α each implement a definition of the *class method* `show`. The type of `show` is written `Show $\alpha \Rightarrow \alpha \rightarrow \text{String}$` : *Given an argument of type α , look in the global listing of `Show` instances, find the one for α , and use that*; if α has no `Show` instance, then we have a type error. One says “the type variable α has is restricted to be a `Show` instance” —as indicated on the left side of the “ \Rightarrow ” symbol.

E.g., for the `List` datatype we defined, we may declare it to be ‘showable’ like so:

```
1 instance Show a => Show (List a) where
2   show Nil      = "Nope, nothing here"
3   show (Cons x xs) = "Saw " ++ show x ++ ", then " ++ show xs
```

That is:

1. If a is showable, then `List a` is also showable.
2. Here’s how to show `Nil` directly.
3. We show `Cons x xs` by using the `show` of a on x , then recursively showing `xs`.

Common Typeclasses	
<code>Show</code>	Show elements as strings, <code>show</code>
<code>Read</code>	How to read element values from strings, <code>read</code>
<code>Eq</code>	Compare elements for equality, <code>==</code>
<code>Num</code>	Use literals 0, 20, ..., and arithmetic <code>+</code> , <code>*</code> , <code>-</code>
<code>Ord</code>	Use comparison relations <code>></code> , <code><</code> , <code>>=</code> , <code><=</code>
<code>Enum</code>	Types that can be listed, <code>[start .. end]</code>
<code>Monoid</code>	Types that model ‘(untyped) composition’
<code>Functor</code>	Type formers that model effectful computation
<code>Applicative</code>	Type formers that can sequence effects
<code>Monad</code>	Type formers that let effects depend on each other

The `Ord` typeclass is declared `class Eq a => Ord a where ...`, so that all ordered types are necessarily also types with equality. One says `Ord` is a *subclass* of `Eq`; and since subclasses *inherit* all functions of a class, we may always replace `(Eq a, Ord a) => ...` by `Ord a => ...`.

You can of-course define your own typeclasses; e.g., the `Monoid` class in Haskell could be defined as follows.

```
class Semigroup a where
  (<>) :: a -> a -> a {- A way to “compose” elements together -}
  {- Axiom: (x <> y) <> z = x <> (y <> z) -}

class Semigroup a => Monoid a where
  mempty :: a {- Axiom: This is a ‘no-op’, identity, for composition <> -}
```

Example monoids $(\alpha, <>, mempty)$ include $(\text{Int}, +, 0)$, $([\alpha], ++, [])$, and (Program statements, sequence “;”, the empty statement) —this last example is approximated as `Term` with ‘let-in’ clauses at the end of this cheatsheet. *Typeclasses are interfaces, possibly with axioms specifying their behaviour.*

As shown earlier, Haskell provides a the *deriving* mechanism for making it easier to define instances of typeclasses, such as `Show`, `Read`, `Eq`, `Ord`, `Enum`. How? Constructor names are printed and read as written as written in the `data` declaration, two values are equal if they are formed by the same construction, one value is less than another if the constructor of the first is declared in the `data` definition before the constructor of the second, and similarly for listing elements out.

Functor

Functors are type formers that “behave” like collections: We can alter their “elements” without messing with the ‘collection structure’ or ‘element positions’. The well-behavedness constraints are called *the functor axioms*.

```
class Functor f where
  fmap :: ( $\alpha \rightarrow \beta$ ) -> f  $\alpha \rightarrow f \beta$ 

(<$>) = fmap {- An infix alias -}
```

The axioms cannot be checked by Haskell, so we can form instances that fail to meet the implicit specifications —two examples are below.

Identity Law: `fmap id = id`

Doing no alteration to the contents of a collection does nothing to the collection.

This ensures that “alterations don’t needlessly mess with element values” e.g., the following is not a functor since it does.

```

{- I probably have an item -}
data Probably a = Chance a Int

instance Functor Probably where
  fmap f (Chance x n) = Chance (f x) (n `div` 2)

```

Fusion Law: $\text{fmap } f \ . \ \text{fmap } g = \text{fmap } (f \ . \ g)$

Reaching into a collection and altering twice is the same as reaching in and altering once.

This ensures that “alterations don’t needlessly mess with collection structure”; e.g., the following is not a functor since it does.

```

import Prelude hiding (Left, Right)

{- I have an item in my left or my right pocket -}
data Pocket a = Left a | Right a

instance Functor Pocket where
  fmap f (Left x) = Right (f x)
  fmap f (Right x) = Left (f x)

```

It is important to note that functors model well-behaved container-like types, but of-course the types do not actually need to contain anything at all! E.g., the following is a valid functor.

```

{- “I totally have an  $\alpha$ -value, it’s either here or there.” Lies! -}
data Liar  $\alpha$  = OverHere Int | OverThere Int

instance Functor Liar where
  fmap f (OverHere n) = OverHere n
  fmap f (OverThere n) = OverThere n

```

Notice that if we altered n , say by dividing it by two, then we break the identity law; and if we swap the constructors, then we break the fusion law. Super neat stuff!

In general, functors take something boring and generally furnish it with ‘coherent’ structure, but **there is not necessarily an α ‘inside’ $f \ \alpha$** . E.g., $f \ \alpha = (\epsilon \rightarrow \alpha)$ has as values “recipes for forming an α -value”, but unless executed, there is no α -value.

-
- ◊ $\text{fmap } f \ xs \approx$ for each element x in the ‘collection’ xs , yield $f \ x$.
 - ◊ Haskell can usually **derive** functor instances since they are **unique**: Only one possible definition of **fmap** will work.
 - ◊ Reading the functor axioms left-to-right, they can be seen as *optimisation laws* that make a program faster by reducing work.
 - ◊ The two laws together say *fmap distributes over composition*:
 $\text{fmap } (f_1 \ . \ f_2 \ . \ \dots \ . \ f_n) = \text{fmap } f_1 \ . \ \dots \ . \ \text{fmap } f_n$ for $n \geq 0$.

Naturality Theorems: If $p :: f \ a \rightarrow g \ a$ for some *functors* f and g , then $\text{fmap } f \ . \ p = p \ . \ \text{fmap } f$ for any *function* f .

Hence, any generic property $p :: f \ \alpha \rightarrow \epsilon$ is invariant over **fmaps**: $p(\text{fmap } f \ xs) = p \ xs$. E.g., the length of a list does not change even when an **fmap** is applied.

Functor Examples

Let f_1, f_2 be functors and ϵ be a given type.

Type Former	$f \ \alpha$	$f \ <\$> \ x$
Identity	α	$f \ <\$> \ x = f \ x$
Constant	ϵ	$f \ <\$> \ x = x$
List	$[\alpha]$	$f \ <\$> \ [x_0, \dots, x_n] = [f \ x_0, \dots, f \ x_n]$
Either	$\text{Either } \epsilon \ \alpha$	$f \ <\$> \ x = \text{right } f$
Product	$(f_1 \ \alpha, f_2 \ \alpha)$	$f \ <\$> \ (x, y) = (f \ <\$> \ x, f \ <\$> \ y)$
Composition	$f_1 \ (f_2 \ \alpha)$	$f \ <\$> \ x = (\text{fmap } f) \ <\$> \ x$
Sum	$\text{Either } (f_1 \ \alpha) \ (f_2 \ \alpha)$	$f \ <\$> \ ea = f \ +++ \ f$
Writer	(ϵ, α)	$f \ <\$> \ (e, x) = (e, f \ x)$
Reader	$\epsilon \rightarrow \alpha$	$f \ <\$> \ g = f \ . \ g$
State	$\epsilon \rightarrow (\epsilon, \alpha)$	$f \ <\$> \ g = \text{second } f \ . \ g$

Notice that **writer** is the product of the constant and the identity functors.

Unlike **reader**, the type former $f \ \alpha = \alpha \rightarrow \epsilon$ is *not* a functor since there is no way to implement **fmap**. In contrast, $f \ \alpha = (\alpha \rightarrow \epsilon, \alpha)$ *does* have an implementation of **fmap**, but it is not lawful.

Applicative

Applicatives are collection-like types that can apply collections of functions to collections of elements.

In particular, *applicatives can fmap over multiple arguments*; e.g., if we try to add `Just 2` and `Just 3`, we find `(+) <$> Just 2 :: Maybe (Int → Int)` and this is not a function and so cannot be applied further to `Just 3` to get `Just 5`. We have both the function and the value wrapped up, so we need a way to apply the former to the latter. The answer is `(+) <$> Just 2 <*> Just 3`.

```
class Functor f => Applicative f where
  pure    :: a -> f a
  (<*>)   :: f (a -> b) -> f a -> f b {- "apply" -}

{- Apply associates to the left: p <*> q <*> r = (p <*> q) <*> r -}
```

The method `pure` lets us inject values, to make ‘singleton collections’.

- ◊ *Functors transform values inside collections; applicatives can additionally combine values inside collections.*
- ◊ Exercise: If α is a monoid, then so too is $f \alpha$ for any applicative f .

The applicative axioms ensure that `apply` behaves like usual functional application:

- ◊ Identity: `pure id <*> x = x` —c.f., `id x = x`
- ◊ Homomorphism: `pure f <*> pure x = pure (f x)` —it really is function application on pure values!
 - Applying a non-effectful function to a non-effectful argument in an effectful context is the same as just applying the function to the argument and then injecting the result into the content.
- ◊ Interchange: `p <*> pure x = pure ($ x) <*> p` —c.f., `f x = ($ x) f`
 - Functions f take x as input \approx Values x project functions f to particular values
 - When there is only one effectful component, then it does not matter whether we evaluate the function first or the argument first, there will still only be one effect.
 - Indeed, this is equivalent to the law: `pure f <*> q = pure (flip ($)) <*> q <*> pure f`.
- ◊ Composition: `pure (.) <*> p <*> q <*> r = p <*> (q <*> r)`
 - c.f., `(f . g) . h = f . (g . h)`.

If we view $f \alpha$ as an “effectful computation on α ”, then the above laws ensure `pure` creates an “effect free” context. E.g., if $f \alpha = [\alpha]$ is considered “nondeterministic α -values”, then `pure` just treats usual α -values as nondeterministic but with no ambiguity, and `fs <*> xs` reads “if we nondeterministically have a choice f from fs , and we nondeterministically an x from xs , then we nondeterministically obtain $f x$.” More concretely, if I’m given randomly addition or multiplication along with the argument 3 and another argument that could be 2, 4, or 6, then the result would be obtained by considering all possible combinations: `[(+), (*)] <*> pure 3 <*> [2, 4, 6] = [5, 7, 9, 6, 12, 18]`. The name “`<*>`” is suggestive of this ‘cartesian product’ nature.

Given a definition of `apply`, the definition of `pure` may be obtained by unfolding the identity axiom.

Using these laws, we regain the original `fmap` —since `fmap`’s are *unique* in Haskell— thereby further cementing that applicatives model “collections that can be functionally applied”: `f <$> x = pure f <*> x`. (Hence, every applicative is a functor whether we like it or not.)

- ◊ The identity applicative law is then just the identity law of functor.
- ◊ The homomorphism law now becomes: `pure . f = fmap f . pure`.
 - This is the “naturality law” for `pure`.

The laws may be interpreted as left-to-right rewrite rules and so are a procedure for transforming any applicative expression into the canonical form of “a pure function applied to effectful arguments”: `pure f <*> x1 <*> ... <*> xn`. In this way, one can compute in-parallel the, necessarily independent, x_i then combine them together.

Notice that the canonical form generalises `fmap` to n -arguments:

Given $f :: \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta$ and $x_i :: f \alpha_i$, we obtain an $(f \beta)$ -value.

The case of $n = 2$ is called `liftA2`, $n = 1$ is just `fmap`, and for $n = 0$ we have `pure`!

Notice that `liftA2` is essentially the cartesian product in the setting of lists, or (`<&>`) below —c.f., `sequenceA :: Applicative f => [f a] -> f [a]`.

```
(<&>) :: f a -> f b -> f (a, b) {- Not a standard name! -}
(<&>) = liftA2 (,) -- i.e., p <&> q = (,) <$> p <*> q
```

This is a pairing operation with properties of `(,)` mirrored at the applicative level:

```
{- Pure Pairing -} pure x <&> pure y = pure (x, y)
{- Naturality -} (f &&& g) <$> (u <&> v) = (f <$> u) <&> (g <$> v)

{- Left Projection -} fst <$> (u <&> pure ()) = u
{- Right Projection -} snd <$> (pure () <&> v) = v
{- Associativity -} assoc1 <$> (u <&> (v <&> w)) = (u <&> v) <&> w
```

The final three laws above suffice to prove the original applicative axioms, and so we may define `p <*> q = uncurry ($) <$> (p <&> q)`.

Applicative Examples

Let f_1, f_2 be functors and let ϵ a type.

Functor	$f \ \alpha$	$f \ \langle * \rangle \ x$
Identity	α	$f \ \langle * \rangle \ x = f \ x$
Constant	ϵ	$e \ \langle * \rangle \ d = e \ \langle \rangle \ d$
List	$[\alpha]$	$fs \ \langle * \rangle \ xs = [f \ x \mid f \leftarrow fs, x \leftarrow xs]$
Either	$\text{Either } \epsilon \ \alpha$	$ef \ \langle * \rangle \ ea = \text{right } (\lambda f \rightarrow \text{right } f \ ea) \ ef$
Composition	$f_1 (f_2 \ \alpha)$	$f \ \langle * \rangle \ x = (\langle * \rangle) \ \langle \$ \rangle \ f \ \langle * \rangle \ x$
Product	$(f_1 \ \alpha, f_2 \ \alpha)$	$(f, g) \ \langle * \rangle \ (x, y) = (f \ \langle * \rangle \ x, g \ \langle * \rangle \ y)$
Sum	$\text{Either } (f_1 \ \alpha) \ (f_2 \ \alpha)$	Challenge: Assume $\eta :: f_1 \ a \rightarrow f_2 \ a$
Writer	(ϵ, α)	$(a, f) \ \langle * \rangle \ (b, x) = (a \ \langle \rangle \ b, f \ x)$
Reader	$\epsilon \rightarrow \alpha$	$f \ \langle * \rangle \ g = \lambda e \rightarrow f \ e \ (g \ e)$ —c.f., SKI
State	$\epsilon \rightarrow (\epsilon, \alpha)$	$sf \ \langle * \rangle \ sa = \lambda e \rightarrow \text{let } (e', f) = sf \ e$ in $\text{second } f \ (sa \ e')$

In the writer and constant cases, we need ϵ to also be a monoid. When ϵ is *not* a monoid, then those two constructions give examples of functors that are *not* applicatives —since there is no way to define `pure`. In contrast, $f \ \alpha = (\alpha \rightarrow \epsilon) \rightarrow \text{Maybe } \epsilon$ is not an applicative since no definition of `apply` is lawful.

Since readers $(\rightarrow) \ r$ are applicatives, we may, for example, write $(\oplus) \ \langle \$ \rangle \ f \ \langle * \rangle \ g$ as a terse alternative to the “pointwise \oplus ” method $\lambda x \rightarrow f \ x \oplus g \ x$. E.g., using $(\&\&)$ gives a simple way to chain predicates.

Do-Notation —Subtle difference between applicatives and monads

Recall the `map` operation on lists, we could define it ourselves:

```
map' :: (α -> β) -> [α] -> [β]
map' f [] = []
map' f (x:xs) = let y = f x
                  ys = map' f xs
                  in (y:ys)
```

If instead the altering function `f` returned effectful results, then we could gather the results along with the effect:

```
{-# LANGUAGE ApplicativeDo #-}

mapA :: Applicative f => (α -> f β) -> [α] -> f [β]
mapA f [] = pure []
mapA f (x:xs) = do y <- f x
                  ys <- mapA f xs
                  pure (y:ys)
{- ≈ (<.>) <$> f x <*> mapA f xs -}
```

Applicative syntax can be a bit hard to write, whereas `do`-notation is more natural and reminiscent of the imperative style used in defining `map'` above. For instance, the intuition that $fs \ \langle * \rangle \ ps$ is a cartesian product is clearer in `do`-notation: $fs \ \langle * \rangle \ ps \approx \text{do } \{f \leftarrow fs; x \leftarrow ps; \text{pure } (f \ x)\}$ where the right side is read “for-each f in fs , and each x in ps , compute $f \ x$ ”.

In-general, $\text{do } \{x_1 \leftarrow p_1; \dots; x_n \leftarrow p_n; \text{pure } e\} \approx \text{pure } (\lambda x_1 \dots x_n \rightarrow e) \ \langle * \rangle \ p_1 \ \langle * \rangle \ \dots \ \langle * \rangle \ p_n$ **provided** p_i does not mention x_j for $j < i$; but e may refer to all x_i . If any p_i mentions an earlier x_j , then we could not translate the `do`-notation into an applicative expression.

If $\text{do } \{x \leftarrow p; y \leftarrow qx; \text{pure } e\}$ has qx being an expression **depending** on x , then we could say this is an abbreviation for $(\lambda x \rightarrow (\lambda y \rightarrow e) \ \langle \$ \rangle \ qx) \ \langle \$ \rangle \ p$ but this is of type $f \ (f \ \beta)$. Hence, to allow later computations to depend on earlier computations, we need a method `join :: f (f α) → f α` with which we define $\text{do } \{x \leftarrow p; y \leftarrow qx; \text{pure } e\} \approx \text{join } \$ \sim (\lambda x \rightarrow (\lambda y \rightarrow e) \ \langle \$ \rangle \ qx) \ \langle \$ \rangle \ p$.

Applicatives with a `join` are called monads and they give us a “programmable semicolon”. Since later items may depend on earlier ones, $\text{do } \{x \leftarrow p; y \leftarrow q; \text{pure } e\}$ could be read “let x be the value of computation p , let y be the value of computation q , then combine the values via expression e ”. Depending on how $\langle * \rangle$ is implemented, such ‘let declarations’ could short-circuit (`Maybe`) or be nondeterministic (`List`) or have other effects such as altering state.

As the `do`-notation clearly shows, the primary difference between `Monad` and `Applicative` is that `Monad` allows dependencies on previous results, whereas `Applicative` does not.

`Do`-syntax also works with tuples and functions —c.f., reader monad below— since they are monadic; e.g., every clause $x \leftarrow f$ in a functional `do`-expression denotes the resulting of applying `f` to the (implicit) input. More concretely:

```
go :: (Show a, Num a) => a -> (a, String)
go = do {x <- (1+); y <- show; return (x, y)}

-- go 3 = (4, "3")
```

Likewise, tuples, lists, etc.

Formal Definition of Do-Notation

For a general applicative f , a `do` expression has the form `do {C; r}`, where C is a (possibly empty) list of commands separated by semicolons, and r is an expression of type $f \beta$, which is also the type of the entire `do` expression. Each command takes the form $x \leftarrow p$, where x is a variable, or possibly a pattern; if $p :: f \alpha$ then $x :: \alpha$. In the particular case of the anonymous variable, $_ \leftarrow p$ may be abbreviated to p .

The translation of a `do` expression into `<*>/join` operations and `where` clauses is governed by three rules—the last one only applies in the setting of a monad.

```
(1) do {r} = r
(2A) do {x ← p; C; r} = q <*> p where q x = do {C; r} --Provided x ∉ C
(2M) do {x ← p; C; r} = join $ map q p where q x = do {C; r}
```

{- Fact: When $x \notin C$, (2A) = (2M). -}

By definition chasing and induction on the number of commands C , we have:

[CollapseLaw] `do {C; do {D; r}}` = `do {C; D; r}`

Likewise:

```
[Map] fmap f p = do {x ← p; pure (f x)} -- By applicative laws
[Join] join ps = do {p ← ps; p} -- By functor laws
```

Do-Notation Laws: Here are some desirable usability properties of `do`-notation.

```
[RightIdentity] do {B; x ← p; pure x} = do {B; p}
[LeftIdentity] do {B; x ← pure e; C; r} = do {B; C[x := e]; r[x := e]}
[Associativity] do {B; x ← do {C; p}; D; r} = do {B; C; x ← p; D; r}
```

Here, B , C , D range over sequences of commands and $C[x := e]$ means the sequence C with all free occurrences of x replaced by e .

- ◊ Associativity gives us a nice way to ‘inline’ other calls.
- ◊ The LeftIdentity law, read right-to-left, lets us “locally give a name” to the possibly complex expression e .

If `pure` forms a singleton collection, then LeftIdentity is a “one-point rule”: We consider *all* $x \leftarrow \text{pure } e$, but there is only *one* such x , namely e !

In the applicative case, where the clauses are independent, we can prove, say, RightIdentity using the identity law for applicatives—which says essentially

`do {x <- p; pure x} = p`—then apply induction on the length of B .

What axioms are needed for the monad case to prove the `do`-notation laws?

Monad Laws

Here is the definition of the monad typeclass.

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

  (<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
  f <=< g = join . fmap f . g
```

Where’s `join`!? Historically, monads entered Haskell first with interface `(>=)`, `return`; later it was realised that `return = pure` and the relationship with applicative was cemented.

‘Bind’ (`>=`) is definable from `join` by `ma >= f = join (fmap f ma)`, and, for this reason, bind is known as “flat map” or “concat map” in particular instances. For instance, the second definition of `do`-notation could be expressed:

```
(2M') do {x ← p; C; r} = p >>= q where q x = do {C; r}
```

Conversely, `join ps = do {p ← ps; p} = ps >= id`. Likewise, with (2M’), note how `(<*>)` can be defined directly in-terms of `(>=)`—c.f., `mf <*> mx = do {f ← mf; x ← mx; return (f x)}`.

Since `fmap f p = do {x ← p; return (f x)} = p >= return . f`, in the past monad did not even have functor as a superclass—c.f., `liftM`.

The properties of `>=`, `return` that prove the desired `do`-notation laws are:

```
[LeftIdentity] return a >>= f ≡ f a
[RightIdentity] m >>= return ≡ m
[Associativity] (m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
               i.e., (m >>= (\x -> f x)) >>= g
                   = m >>= (\x -> f x >>= g)
```

Equivalently, show the ‘fish’ (\leq) is associative with identity being `pure` —c.f., monoids!

It is pretty awesome that `(>=)`, `return` give us a functor, an applicative, and (dependent) `do`-notation! Why? Because `bind` does both the work of `fmap` and `join`. Thus, `pure`, `fmap`, `join` suffice to characterise a monad.

Join determines how a monad behaves!

The monad laws can be expressed in terms of `join` directly:

[Associativity] `join . fmap join = join . join`
{- The only two ways to get from “m (m (m α))” to “m α” are the same. -}

[Identity Laws] `join. fmap pure = join . pure = id`
{- Wrapping up “m α” gives an “m (m α)” which flattens to the original element. -}

Then, notice that the (free) naturality of `join` is:

`join . fmap (fmap f) = fmap f . join :: m (m α) → m β`

Again, note that `join` doesn’t merely flatten a monad value, but rather performs the necessary logic that determines *how the monad behaves*.

E.g., suppose $m\ \alpha = \epsilon \rightarrow (\epsilon, \alpha)$ is the type of α -values that can be configured according to a fixed environment type ϵ , along with the possibly updated configuration —i.e., functions $\epsilon \rightarrow (\epsilon, \alpha)$. Then any $a : \epsilon \rightarrow (\epsilon, \epsilon \rightarrow (\epsilon, \alpha))$ in $m\ (m\ \alpha)$ can be considered an element of $m\ \alpha$ if we *propagate the environment configuration* through the outer layer to obtain a new configuration for the inner layer: $\lambda e \rightarrow \text{let } (e', a') = a\ e \text{ in } a'\ e'$. The `join` dictates how a configuration is *modified then passed along*: We have two actions, a and a' , and `join` has *sequenced* them by pushing the environment through the first thereby modifying it then pushing it through the second.

Monad Examples

Let f_1, f_2 be functors and let ϵ a type.

Applicative	$m\ \alpha$	<code>join :: m (m α) → m α</code>
Identity	α	$\lambda x \rightarrow x$
Constant	ϵ	$\lambda x \rightarrow x$ —Shucks! Not a monad!
List	$[\alpha]$	$\lambda xss \rightarrow \text{foldr } (++)\ []\ xss$
Either	$\text{Either } \epsilon\ \alpha$	Exercise $\hat{\quad}\hat{\quad}$
Composition	$f_1\ (f_2\ \alpha)$	Nope! Not a monad!
Product	$(f_1\ \alpha, f_2\ \alpha)$	$\lambda p \rightarrow (\text{fst } <\$>\ p, \text{snd } <\$>\ p)$
Writer	(ϵ, α)	$\lambda (e, (e', a)) \rightarrow (e <> e', a)$
Reader	$\epsilon \rightarrow \alpha$	$\lambda ra \rightarrow \lambda e \rightarrow ra\ e\ e$
State	$\epsilon \rightarrow (\epsilon, \alpha)$	$\lambda ra \rightarrow \lambda e \rightarrow \text{let } (e', a) = ra\ e \text{ in } a\ e'$

In `writer`, we need ϵ to be a monoid.

- ◊ Notice how, in `writer`, `join` merges the outer context with the inner context: *Sequential writes are mappended together!*
- ◊ If `pure` forms ‘singleton containers’ then `join` flattens containers of containers into a single container.

Excluding the trivial monoid, the constant functor is *not* a monad: It fails the monad identity laws for `join`. Similarly, $f\ \alpha = \text{Maybe } (\alpha, \alpha)$ is an applicative but *not* a monad —since there is no lawful definition of `join`. Hence, applicatives are strictly more generally than monads.

Running Example —A Simple Arithmetic Language

Let’s start with a weak language:

```
data Term = Int Int | Div Term Term deriving Show

thirteen = Int 1729 'Div' (Int 133 'Div' Int 1)
boom      = Int 1729 'Div' (Int 12 'Div' Int 0)

eval0 :: Term -> Int
eval0 (Int n) = n
eval0 (n 'Div' d) = let top    = eval0 n
                    bottom = eval0 d
                    in top 'div' bottom
```

How do we accomodate safe division by zero? Print to the user what’s happening at each step of the calculation? Have terms that access ‘global’ variables? Have terms that can store named expressions then access them later?

We’ll make such languages and their `eval`’s will be nearly just as simple as this one (!) but accomodate these other issues.

Maybe —Possibly Failing Computations

Safe evaluator: No division errors.

```
eval1 :: Term -> Maybe Int
eval1 (Int n) = pure n
eval1 (n 'Div' d) = do t <- eval1 n
                      b <- eval1 d
                      if b == 0 then Nothing else pure (t 'div' b)
```

Exercise: Rewrite eval_i without `do`-notation and you'll end-up with nested case analysis leading into a straicase of code that runs right off the page.

◊ Applicative is enough for eval_1 , eval_2 , eval_3 , but eval_4 needs `Monad`.

Writer —Logging Information as we Compute

Use a pair type $W \in \alpha$ to keep track of an environment ϵ and a value α .

```
data Writer  $\in \alpha$  = W  $\in \alpha$  deriving Show
```

```
write ::  $\epsilon$  -> Writer  $\in$  ()
write e = W e ()
```

```
instance Functor (Writer  $\epsilon$ ) where
  fmap f (W e a) = W e (f a)
```

Aggregate, merge, environments using their monoidal operation.

```
instance Monoid  $\epsilon$  => Applicative (Writer  $\epsilon$ ) where
  pure a = W mempty a
  (W e f) <*> (W d a) = W (e <> d) (f a)
```

```
instance Monoid  $\epsilon$  => Monad (Writer  $\epsilon$ ) where
  (>>=) = \ ma f -> join (pure f <*> ma)
  where join (W e (W d a)) = W (e <> d) a
```

An evaluator that prints to the user what's going on.

```
eval2 :: Term -> Writer String Int
eval2 it@(Int n) = W ("\n Evaluating: " ++ show it) n
eval2 it@(n 'Div' d) = do write $ "\n Evaluating: " ++ show it
                          t <- eval2 n
                          b <- eval2 d
                          pure $ (t 'div' b)
```

```
-- Try this! With "boom", we get to see up to the boint of the error ^_^
-- let W e x = eval2 thirteen in putStrLn e
```

Reader —Accessing 'Global, read-only, data'

Use a function type $\epsilon \rightarrow \alpha$ to get α -values that 'reads' from a configuration environment ϵ .

```
data Reader  $\in \alpha$  = R {run ::  $\epsilon$  ->  $\alpha$ }
```

```
instance Functor (Reader  $\epsilon$ ) where
  fmap f (R g) = R $ f . g
```

```
instance Applicative (Reader  $\epsilon$ ) where
  pure a = R $ const a
  (R f) <*> (R g) = R $ \e -> f e (g e) {- "S" combinator -}
```

```
instance Monad (Reader  $\epsilon$ ) where
  ma >>= f = join (pure f <*> ma)
  where join (R rf) = R $ \e -> run (rf e) e
```

A language with access to global variables; uninitialised variables are 0 by default.

```
data Term = Int Int | Div Term Term | Var String deriving Show
```

```
type GlobalVars = [(String, Int)]
```

```
valuefrom :: String -> GlobalVars -> Int
valuefrom x gvs = maybe 0 id $ lookup x gvs
```

```

eval3 :: Term -> Reader GlobalVars Int
eval3 (Int x) = pure x
eval3 (Var x) = R $ \e -> x 'valuefrom' e
eval3 (n 'Div' d) = do t <- eval3 n
                      b <- eval3 d
                      pure (t 'div' b)

state = [("x", 1729), ("y", 133)] :: GlobalVars
thirteen = Var "x" 'Div' (Var "y" 'Div' Int 1)
-- run (eval3 thirteen) state

```

State —Read and write to local storage

Let's combine writer and reader to get state: We can both read and write to data by using functions $\epsilon \rightarrow (\epsilon, \alpha)$ that read from an environment ϵ and result in a new environment as well as a value.

◊ $\text{IO } \alpha \cong \text{State TheRealWorld } \alpha$;-)

```

data State  $\epsilon$   $\alpha$  = S {run ::  $\epsilon \rightarrow (\epsilon, \alpha)$ }

push :: Monoid  $\epsilon \Rightarrow \epsilon \rightarrow \text{State } \epsilon ()$ 
push d = S $ \e -> (d <> e, ())

instance Functor (State  $\epsilon$ ) where
  fmap f (S g) = S $ \e -> let (e', a) = g e in (e', f a)

instance Applicative (State  $\epsilon$ ) where
  pure a = S $ \e -> (e, a)
  (S sf) <*> (S g) = S $ \e -> let (e', a) = g e
                                (e'', f) = sf e' in (e'', f a)

instance Monad (State  $\epsilon$ ) where
  ma >>= f = join (pure f <*> ma)
  where join (S sf) = S $ \e -> let (e', S f) = sf e in f e'

```

A simple language with storage; a program's value is the value of its final store.

```

data Expr = Let String Expr Expr | Var String | Int Int | Div Expr Expr
           deriving Show

eval4 :: Expr -> State GlobalVars Int
eval4 (Var x) = S $ \e -> let r = x 'valuefrom' e in ((x,r):e, r)
eval4 (Int x) = pure x
eval4 (Let x t body) = do n <- eval4 t
                          push [(x, n)] -- Applicative is NOT enough here!
                          eval4 body
eval4 (n 'Div' d) = do t <- eval4 n; b <- eval4 d; pure (t 'div' b)

thirteen = Let "x" (Int 1729)
           $ Let "y" (Int 133 'Div' Int 1)
           $ Var "x" 'Div' Var "y"

-- run (eval4 thirteen) []

```

Exercise: Add to the original `Term` type a constructor `Rndm [Term]`, where `Rndm [t1, ..., tn]` denotes non-deterministically choosing one of the terms `ti`. Then write an evaluator that considers all possible branches of a computation: `eval5 : Term → [Int]`.

If we want to mixin any of the features for our evaluators, we need to use 'monad transformers' since monads do not compose in general.

Reads

- ◊ *Introduction to Functional Programming* by Richard Bird
 - Assuming no programming, this book end by showing how to write a theorem prover powerful enough to prove many of laws scattered throughout the book.
- ◊ *Monads for functional programming* by Philip Wadler
 - This covers the `evali` and more `^_`
- ◊ *Comprehending Monads* by Philip Wadler
- ◊ *What I Wish I Knew When Learning Haskell*
- ◊ *Typeclassopedia* —*The essentials of each type class are introduced, with examples, commentary, and extensive references for further reading.*

- ◊ [You Could Have Invented Monads! \(And Maybe You Already Have.\)](#)
- ◊ [Learn You a Haskell for Great Good](#) —An accessible read with many examples, and drawings
- ◊ [The Haskell WikiBook](#) —Has four beginner’s tracks and four advanced tracks
- ◊ [Category Theory Cheat Sheet](#) —The “theory of typed composition”: Products, Sums, Functors, Natural Transformations ^ _ ^
- ◊ [Agda Cheat Sheet](#) —Agda is Haskell on steroids in that it you can invoke Haskell code and write proofs for it.
- ◊ [LINQ for applicatives and monads](#).
 - $\text{Monads} \approx \text{SQL/Linq} \approx \text{Comprehensions/Generators}$