Folktale Documentation

Release 1.0

Quildreen Motta

Nov 05, 2017

Contents

Folktale is a suite of libraries for generic functional programming in JavaScript that allows you to write elegant modular applications with fewer bugs, and more reuse.

CHAPTER 1

Guides

- ***Getting Started*** A series of quick tutorials to get you up and running quickly with the Folktale libraries.
- ***API reference*** A quick reference of Folktale's libraries, including usage examples and cross-references.

CHAPTER 2

Indices and tables

- **Global Module Index** Quick access to all modules.
- **General Index** All functions, classes, terms, sections.
- **Search page** Search this documentation.

CHAPTER 3

Other resources

- Licence information

3.1 Getting started

This guide will cover everything you need to start using the Folktale project right away, from giving you a brief overview of the project, to installing it, to creating a simple example. Once you get the hang of things, the *Folktale By Example* guide should help you understanding the concepts behind the library, and mapping them to real use cases.

3.1.1 So, what's Folktale anyways?

Folktale is a suite of libraries for allowing a particular style of functional programming in JavaScript. This style uses overtly generic abstractions to provide the maximum amount of reuse, and composition, such that large projects can be built in a manageable way by just gluing small projects together. Since the concepts underneath the library are too generic, however, one might find it difficult to see the applications of the data structures to the problems they need to solve (which the *Folktale By Example* guide tries to alleviate by using real world examples and use cases to motivate the concepts). However, once these concepts are understood, you open up a world of possibilities and abstractive power that is hard to find anywhere else.

The main goal of Folktale is to allow the development of robust, modular, and reusable JavaScript components easier, by providing generic primitives and powerful combinators.

3.1.2 Do I need to know advanced maths?

Short answer is **no**. You absolutely don't need to know any sort of advanced mathematics to use the Folktale libraries.

That said, most of the concepts used by the library are derived from Category Theory, Algebra, and other fields in mathematics, so the if you want to extrapolate from the common use cases, and create new abstractions, it will help you tremendously to be familiar with these branches of mathematics.

3.1.3 Okay, how can I use it?

Good, let's get down to the good parts!

Folktale uses a fairly modular structure, where each library is provided as a separate package. To manage all of them, we use NPM. If you're already using Node, you're all set, just skip to the next section.

If you're not using Node, you'll need to install it so you can grab the libraries. Don't worry, installing Node is pretty easy:

1. Go the the Node.js download page.
2. If you're on Windows, grab the `.msi` installer. If you're on Mac, grab the `.pkg` installer.

Note: If you're on Linux, the easiest way is to grab the **Linux Binaries**, extract them to some folder, and place the `node` and `npm` binaries on your `$PATH`

```
~ mkdir ~/Applications/node-js
~ cd ~/Applications/node-js
~ wget http://nodejs.org/dist/v0.10.24/node-v0.10.24-linux-x64.tar.gz
# or linux-x86.tar.gz, for 32bit architectures
~ tar -xzf node-.tar.gz
~ cd /usr/local/bin
~ sudo ln -s ~/Applications/node-js/node-v0.10.24-linux-x64/bin/node node
~ sudo ln -s ~/Applications/node-js/node-v0.10.24-linux-x64/bin/npm npm
```

On Ubuntu, you can also use Chris Lea's PPA.

3.1.4 Hello, world.

Now that you have Node, we can get down to actually using the library. For this, let's create a new directory where we'll install the library:

```
~ mkdir ~/folktale-hello-world
~ cd ~/folktale-hello-world
~ npm install data.maybe
```

The `npm install` command will grab the library for you. In this case, the library is `data.maybe`, which provides a data structure for modelling values that might not exist (like nulls, but safer). It should only take a few seconds to get everything installed, and if all goes well, you'll have a `node_modules` folder with all the stuff.

Now, run `node` to get dropped into a Read-Eval-Print-Loop, which will allow us to play around with the library interactively. Once in the REPL, you can load the library:

```
// We load the library by "require"-ing it
var Maybe = require('data.maybe')

// Returns Maybe.Just(x) if some `x` passes the predicate test
// Otherwise returns Maybe.Nothing()
function find(predicate, xs) {
  return xs.reduce(function(result, x) {
    return result.orElse(function() {
      return predicate(x)?    Maybe.Just(x)
      :      /* otherwise */  Maybe.Nothing()
    })
  }, Maybe.Nothing())
```

```
}


var numbers = [1, 2, 3, 4, 5]

var anyGreaterThan2 = find(function(a) { return a > 2 }, numbers)
// => Maybe.Just(3)

var anyGreaterThan8 = find(function(a) { return a > 8 }, numbers)
// => Maybe.Nothing
```

3.1.5 What about the Browser?

Running in the browser takes just a little bit more of effort. To do so, you'll first need the Browserify too, which converts modules using the Node format, to something that the Browsers can use. Browserify is just an NPM module, so it's easy to get it:

```
$ npm install browserify
```

Since Browserify has quite a bit more of dependencies than our `data.maybe` library, it'll take a few seconds to fully install it. Once you've got Browserify installed, you'll want to create your module using the Node format. So, create a `hello.js` with the following content:

```
// We load the data.maybe library, just like in Node
var Maybe = require('data.maybe')

Maybe.Just("Hello, world!").chain(function(value) {
  document.body.appendChild(document.createTextNode(value))
})
```

To compile this file with Browserify, you run the Browserify command giving the file as input:

```
~ $(npm bin)/browserify hello.js > bundle.js
```

And finally, include the `bundle.js` file in your webpage:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello, World</title>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

By opening the page on your webbrowser, you should see `Hello, World!` added to the page.

3.1.6 What else do I get?

Folktale is a large collection of base libraries, and still largely a work in progress, but there's a lot that is already done and can be used today!

- Safe optional value (replaces nullable types) with the Maybe structure.

- Disjunction type (commonly encodes errors) with the Either structure.
- Disjunction with failure aggregation with the Validation structure.
- Asynchronous values and computations with the Task structure.
- Common and useful combinators from Lambda Calculus.
- Common and useful monadic combinators.

Each of them are fairly broad concepts. The recommended way of getting familiar with them is working through the *Folktale By Example* guide, which will explain each concept through a series of real world use cases.

3.2 Folktale by Example

> **Warning: This book is a work in progress!**
>
> This is largely a draft at this point, so if you see any problems, feel free to file a ticket and I'll get to fixing it up asap.

Category Theory is a relatively new branch of mathematics with fairly abstract concepts. Functional programming libraries use such concepts for maximising their abstraction power and general usefulness, but it comes with a certain drawback: most of the constructions provide little or no guidance for concrete use cases. This is a problem in particular with newcomers to this style of programming, who often find themselves lost, asking questions like "But why are Monads useful?"

In this book, you will walk through concrete applications of concepts in Category Theory, Abstract Algebra, and other branches of mathematics used by functional programming, as well as concepts from functional programming itself. By looking at concrete instances of these concepts, you can build a mental framework for generalising and abstracting problems in a way that makes your code more reusable, and more robust.

Note: Do note that this isn't a book *about* Category Theory or any other mathematical field, the concepts presented in this book are just influenced by them.

3.2.1 Approach

People tend to have a fairly difficult time reasoning about abstractions, but they can easily recognise concrete instances of those abstractions. With enough examples, they can then build their own mental model of that abstraction and, having that mental model, they'll be able to apply that generalisation to find other concrete instances of that abstractions on their own.

With that in mind, this book tries to present its readers with concrete applications of a concept before discussing the concept itself. It does so by presenting, at each chapter, a set of related problems, discussing concrete solutions for those problems, and finally extrapolating to a general solution that captures the *pattern* in those concrete solutions.

3.2.2 Who should read this book?

This book is aimed at intermediate and advanced JavaScript programmers who want to take advantage of mathematical concepts to make their JavaScript code bases simpler, more robust and reusable.

You're expected to be comfortable not only with the syntax and basic concepts of the JavaScript language, but also with concepts such as **higher-order programming**, **first-class functions**, **objects**, **prototypes**, and **dynamic dispatch**,

which are going to be the basis for the concepts discussed in this book. Non-JavaScript programmers familiar with those concepts might be able to translate the concepts to their languages, with some work, but a different book might be better suited for their needs.

To make the most out of this book, you'll also need some school-level mathematical reasoning skills, since the generalisation of the concepts will be presented as mathematical laws. Properly understanding them will take some knowledge of equality, substitutability and unification. Albert Y. C. Lai has described the prerequisite mathematical skills for functional programming on a web page.

3.2.3 How is this book organised?

The book is split into a few sections, each section starts with a description its theme, prerequisites and motivation, spends a few chapters talking about concrete examples inside that theme, and concludes with a summary of the abstractions presented. Sections build on top of each other, so it might be difficult to read the book in a non-linear way.

Section 1 discusses functions, and how composition can be used to build new functionality from existing one easily, as well as how JavaScript function affect composition in JS. It presents the `core.lambda`, `core.arity` and `core.operator` libraries.

Section 2 discusses data structures and their transformations. It talks about concepts such as *Functors* and recursion schemes such as *Catamorphisms*. This section also gives a general idea of how *sum types* are modelled in **Folktale**. It presents the `data.maybe`, `data.either` and `data.validation` libraries.

Section 3 discusses advanced transformations of data structures. It talks about concepts such as *Applicative Functors* and *Monads*. It presents new facets of the libraries presented in **Section 2**.

Section 4 discusses approaches to asynchronous concurrency when dealing with simple values. It revisits *Monads*, and talks about new concepts such as *Continuation-Passing style*, *Tasks*, and *Futures*. It presents the `data.task` library.

Section 5 discusses advanced approaches to dealing with multi-value concurrency. It expands on **Section 4** by presenting new concepts such as *back-pressure*, *Signals* and *Channels*. It presents the `data.channel` and `data.signal` libraries.

Section 6 discusses data validation and normalisation in more detail. It presents the `data.validation` and `core.check` libraries.

3.2.4 Examples

The book contains a heavy amount of examples, all of which can be found in the Folktale GitHub repository, under the `docs/examples` folder.

3.2.5 Table of Contents

Section I: Composition

> We are about to study the idea of a computational process. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.
>
> —G. J. Sussman, H. Abelson and J. Sussman in Structure and Interpretation of Computer Programs

In programming we solve problems by breaking them into smaller parts, and then putting such smaller parts back together somehow to construct the whole solution. The same is largely true in *functional* programming. We break problems down into smaller problems, solve them, and put them back together to solve the large one.

Because of this, most of the emphasis in *functional* programming is in **composition**. More concretely, functional programs try to find the right data structures to represent the problem, and then figure out how to transform such data structures from the original problem into the proposed solution. Given that transformations from the problem to the solution are usually too big, functional programs break these transformations into smaller transformations, and even smaller transformations, and then build a big transformation by using the smaller ones.

Transformations in functional programming are captured by, as one would probably have guessed, *functions*. And the act of putting such functions back together to form bigger things is called *function composition*. This section will discuss how composition helps writing better JavaScript programs, and how one can use the Folktale libraries for that.

3.3 API Reference

Folktale is a suite of libraries for generic functional programming in JavaScript. It allows the construction of elegant, and robust programs, with highly reusable abstractions to keep the code base maintainable.

The library is organised by a variety of modules split into logical categories, with the conventional naming of `<Category>.<Module>`. This page provides reference documentation for all the modules in the Folktale library, including usage examples and cross-references for helping you find related concepts that might map better to a particular problem.

3.3.1 Core

Provides the most basic and essential building blocks and compositional operations, which are likely to be used by most programs.

- ***core.arity*** Restricts the arity of variadic functions.
- ***core.check*** Run-time interface checking/contracts for JavaScript values.
- ***core.inspect*** Human-readable representations of built-in and custom objects.
- ***core.lambda*** Essential functional combinators and higher-order functions derived from λ-Calculus.
- ***core.operators*** Curried and first-class versions of JavaScript built-in operators.

Module: `core.arity`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/core.arity/issues

Version 1.0.0

Repository https://github.com/folktale/core.arity

Portability Portable

npm package `core.arity`

Restricts the arity of variadic functions.

Loading

Require the `core.arity` package, after installing it:

```
var arity = require('core.arity')
```

Why?

Since all functions in JavaScript are variadic, programmers often take advantage of this fact by providing more arguments than what a function takes, and the callee can just ignore them. With curried functions, calling a binary function with three arguments ends up invoking the return value of the function with the extra argument!

```
var curry = require('core.lambda').curry;

function add(a, b) {
  return a + b;
}

var cadd = curry(2, add);

cadd(1)(2)     // => 3
cadd(1, 2)     // => 3
cadd(1, 2, 4) // => Error: 3 is not a function
```

To fix this, one would need to wrap the curried function such that the wrapper only passes two arguments to it, and ignores the additional ones:

```
var binary = require('core.arity').binary;

binary(cadd)(1, 2, 4) // => 3
```

Uncategorised

nullary()

core.arity.**nullary**(*f*)

Returns A function that takes no arguments.

$(\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\text{Unit} \to \beta)$

Restricts a variadic function to a nullary one.

unary()

core.arity.**unary**(*f*)

Returns A function that takes one argument.

$(\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\alpha_1 \to \beta)$

Restricts a variadic function to an unary one.

binary()

`core.arity.binary(f)`

Returns A function that takes two arguments.

$(\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\alpha_1 \to \alpha_2 \to \beta)$

Restricts a variadic function to a binary one.

ternary()

`core.arity.ternary(f)`

Returns A function that takes three arguments.

$(\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\alpha_1 \to \alpha_2 \to \alpha_3 \to \beta)$

Restricts a variadic function to a ternary one.

Module: `core.check`

Stability 1 - Experimental

Bug Tracker https://github.com/folktale/core.check/issues

Version 0.1.0

Repository https://github.com/folktale/core.check

Portability Portable

npm package `core.check`

Interface checking for JS values.

Loading

Require the `core.check` package, after installing it:

```
var check = require('core.check')
```

Why?

JavaScript is an untyped language, and this makes it fairly flexible for certain things. More often than not, however, you want to make sure that the values going into a certain code path have some kind of structure, to reduce the complexity of the whole program. `core.check` helps you to do this by providing composable contracts:

```
check.assert(check.String(1))
// => Error: Expected 1 to have tag String

check.assert(check.Or([check.String, check.Boolean])(1))
// => Error: Expected 1 to have tag String, or 1 to have tag Boolean
```

`core.check` can also be used for validating data structures without crashing the process. All contracts return a `Validation(Violation, α)` result. One can then use the *`cata()`* operation on the *`data.validation.Validation`* object to deal with the result of the operation:

```
function logString(a) {
  return check.String(a).cata({
    Failure: function(error){ return 'Not a string: ' + a },
    Success: function(value){ return 'String: ' + value }
  })
}

logString(1)
// => 'Not a string: 1'

logString('foo')
// => 'String: foo'
```

Dependent validations

Value()

`core.check.`**`Value`**(*expected*)

```
α → (α → Validation(Violation, α))
```

An interface that matches the given value by structural equality.

Identity()

`core.check.`**`Identity`**(*expected*)

```
α → (α → Validation(Violation, α))
```

An interface that matches the given value by reference equality.

Higher-order validations

Or()

`core.check.`**`Or`**(*interfaces*)

Returns An interface that matches any of the given interfaces.

```
Array(α → Validation(Violation, α)) → α → Validation(Violation, α)
```

And()

`core.check.`**`And`**(*interfaces*)

Returns An interface that matches only if all of the given interfaces match.

```
Array(α → Validation(Violation, α)) → α → Validation(Violation, α)
```

Seq()

core.check.**Seq**(*interfaces*)

Returns An interface that matches an N-Tuple with the given interfaces.

```
Array( α₁ → Validation<Violation, α₁)
     , α₂ → Validation(Violation, α₂)
     , ...
     , α → Validation(Violation, α)>
→ Array(α₁, α₂, ..., α)
→ Validation(Violation, Array(α₁, α₂, ..., α))
```

ArrayOf()

core.check.**ArrayOf**(*interface*)

Returns An interface that matches an Array with values matching the given interface.

```
(α → Validation(Violation, α)) → α → Validation(Violation, α)
```

ObjectOf()

core.check.**ObjectOf**(*aPattern*)

Returns An interface that matches an Object with the exact key/type mapping given.

```
Object(Validation(Violation, Any)) → Object(Any) → Validation(Violation,␣
↪Object(Any))
```

Primitive validations

Null()

core.check.**Null**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `null` values.

Undefined()

core.check.**Undefined**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `undefined` values.

Boolean()

`core.check.`**`Boolean`**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `Boolean` values.

Number()

`core.check.`**`Number`**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `Number` values.

String()

`core.check.`**`String`**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `String` values.

Function()

`core.check.`**`Function`**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `Function` values.

Array()

`core.check.`**`Array`**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `Array` values.

Object()

core.check.**Object**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches only `Object` values.

Any()

core.check.**Any**(*aValue*)

```
Any → Validation(Violation, Any)
```

An interface that matches any values.

Types and structures

Violation

class core.check.**Violation**

```
type Violation = Tag(String, Any)
               | Equality(Any, Any)
               | Identity(Any, Any)
               | Any(Array(Any))
               | All(Array(Any))

implements
  Equality, Extractor, Reflect, Cata, Semigroup, ToString
```

Represents a violation of an interface's constraint.

+

Validating interfaces

assert()

core.check.**assert**(*aValidation*)

Returns The value, if no violations exist.

Raises

- **TypeError** - If any violation exists.

```
Validation(Violation, α) → α :: throws
```

Type: Violation

class core.check.**Violation**

```
type Violation = Tag(String, Any)
               | Equality(Any, Any)
               | Identity(Any, Any)
               | Any(Array(Any))
               | All(Array(Any))

implements
  Equality, Extractor, Reflect, Cata, Semigroup, ToString
```

Represents a violation of an interface's constraint.

Combining

#concat()

Violation.prototype.**concat**(*aViolation*)

Returns A Violation with the contents combined.

```
@Violation => Violation → Boolean
```

Combines the contents of two Violations.

Comparison and testing

#isTag

Violation.prototype.**isTag**

```
@Violation => Boolean
```

`true` is the Violation has a `Tag` tag.

#isEquality

Violation.prototype.**isEquality**

```
@Violation => Boolean
```

`true` is the Violation has an `Equality` tag.

#isIdentity

`Violation.prototype.`**`isIdentity`**

```
@Violation => Boolean
```

`true` is the Violation has an `Identity` tag.

#isAny

`Violation.prototype.`**`isAny`**

```
@Violation => Boolean
```

`true` is the Violation has an `Any` tag.

#isAll

`Violation.prototype.`**`isAll`**

```
@Violation => Boolean
```

`true` is the Violation has an `All` tag.

#equals()

`Violation.prototype.`**`equals`**`(`*aViolation*`)`

Returns `true` if both Violations have the same contents (by reference equality).

```
@Violation => Violation → Boolean
```

Converting

#toString()

`Violation.prototype.`**`toString`**`()`

Returns A textual representation of the Violation.

```
@Violation => Violation → Boolean
```

Transforming

#cata()

`Violation.prototype.`**`cata`**`(`*aPattern*`)`

Returns The result of applying the right transformation to the Violation.

```
@Violation => { r | Pattern } → β
where type Pattern {
  Tag: (String, Any) → β,
  Equality: (Any, Any) → β,
  Identity: (Any, Any) → β,
  Any: Array(Any) → β,
  All: Array(Any) → β
}
```

Provides a crude form of pattern matching over the Violation ADT. Since Violation also implements the Extractor interface, you may choose to use the Sparkler Sweet.js macro instead for a more powerful form of pattern matching.

Module: `core.inspect`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/core.inspect/issues

Version 1.0.3

Repository https://github.com/folktale/core.inspect

Portability Portable

npm package `core.inspect`

```
Any → String
```

Human-readable representations for built-in and custom objects.

Loading

Require the `core.inspect` package, after installing it:

```
var inspect = require('core.inspect')
```

The module itself is a specialised form of *`core.inspect.show()`* that has a limited `maxDepth`:

```
inspect([1, [2, [3, [4, [5, [6]]]]]])
// => '[1, [2, [3, [4, [5, (...)]]]]]'
```

Why?

Some objects provide a custom representation, some do not. You usually want to see the custom textual representation if an object has it, since just showing its own properties might not give you enough information about it, or might not be as easy to read. But you also want to represent objects that don't have a custom representation as something more useful than `[object Object]`. `core.inspect` solves this problem.

Consider a simple custom type representing a point in a 2d plane:

```
function Point2d(x, y) {
  this.x = x;
  this.y = y;
}

Point2d.prototype.toString = function() {
  return 'Point2d(' + this.x + ', ' + this.y + ')'
}
```

If one wants to print a textual representation of this type, they'd call `Point2d.toString()`:

```
var p1 = new Point2d(10, 20);
p1.toString()
// => (String) "Point2d(10, 20)"
```

But what if you don't know if the object you're dealing with has a custom textual representation or not? In that case, you'd usually try to just display its properties:

```
var Maybe = require('data.maybe');

var player = {
  lastPosition: Maybe.Nothing(),
  currentPosition: Maybe.Just(new Point2d(10, 20))
}

player

// => {
//       "lastPosition": {},
//       "currentPosition": {
//         "value": {
//           "x": 10,
//           "y": 20
//         }
//       }
//     }
```

In this example we have no way of knowing that `lastPosition` contains a `Maybe.Nothing` value, or that `currentPosition` is wrapped in a `Maybe.Just`. A more informative description would be what `core.inspect` gives you:

```
var show = require('core.inspect');

show(player);
// => '{"lastPosition": Maybe.Nothing, "currentPosition": Maybe.Just(Point2d(10, 20))}
↪'
```

Uncategorised

show()

`core.inspect.`**`show`**(*maxDepth*, *value*)

Returns A human-readable representation of the value.

```
Number → Any → String
```

Provides a human-readable representation of built-in values, and custom values implementing the `ToString` interface.

Module: `core.lambda`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/core.lambda/issues

Version 1.0.0

Repository https://github.com/folktale/core.lambda

Portability Portable

npm package `core.lambda`

Core combinators and higher-order functions.

Loading

Require the `core.lambda` package, after installing it:

```
var lambda = require('core.lambda')
```

Why?

Functional programming places heavy emphasis in composition (specially function composition), but JavaScript lacks built-in functionality for composing and transforming functions in order to compose them. `core.lambda` fills this gap by providing tools for composing functions, altering the shape of a function in order to compose them in different ways, and currying/uncurrying.

Uncategorised

identity()

`core.lambda.`**`identity`**(a)

Returns The argument it's given.

$$\alpha \to \alpha$$

The identity combinator. Always returns the argument it's given.

+

constant()

`core.lambda.`**`constant`**(a, b)

Returns The first argument it's given.

$$\alpha \to \beta \to \alpha$$

The constant combinator. Always returns the first argument it's given.

+

apply()

`core.lambda.`**`apply`**(*f*, *a*)

Returns The result of applying `f` to `a`.

$$(\alpha \to \beta) \to \alpha \to \beta$$

Applies a function to an argument.

+

flip()

`core.lambda.`**`flip`**(*f*)

Returns The function `f` with parameters inverted.

$$(\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$$

Inverts the order of the parameters of a binary function.

+

compose()

`core.lambda.`**`compose`**(*f*, *g*)

Returns A composition of `f` and `g`.

$$(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Composes two functions together.

+

curry()

`core.lambda.`**`curry`**(*n*, *f*)

Returns A curried version of `f`, up to `n` arguments.

$$\texttt{:Number} \to (\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta)$$

Transforms any function on tuples into a curried function.

+

spread()

`core.lambda.`**`spread`**(*f*, *xs*)

Returns The result of applying the function `f` to arguments `xs`.

$(\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta) \to (\#[\alpha_1, \alpha_2, \ldots, \alpha] \to \beta)$

Applies a list of arguments to a curried function.

+

uncurry()

`core.lambda.`**`uncurry`**(*f*)

Returns A function on tuples.

$(\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta) \to (\alpha_1, \alpha_2, \ldots, \alpha \to \beta)$

Transforms a curried function into a function on tuples.

+

upon()

`core.lambda.`**`upon`**(*f*, *g*)

Returns A binary function `f` with arguments transformed by `g`.

$(\beta \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \alpha \to \gamma)$

Applies an unary function to both arguments of a binary function.

+

Function: `identity`

`core.lambda.`**`identity`**(*a*)

Returns The argument it's given.

$\alpha \to \alpha$

The identity combinator. Always returns the argument it's given.

Examples

```
identity(3)           // => 3
identity([1])         // => [1]
```

Function: `constant`

`core.lambda.`**`constant`**`(a, b)`

Returns The first argument it's given.

$$\alpha \to \beta \to \alpha$$

The constant combinator. Always returns the first argument it's given.

Examples

```
constant(3)(2)                // => 3
constant('foo')([1])          // => 'foo'
```

Function: `apply`

`core.lambda.`**`apply`**`(f, a)`

Returns The result of applying `f` to `a`.

$$(\alpha \to \beta) \to \alpha \to \beta$$

Applies a function to an argument.

Examples

```
var inc = function(a){ return a + 1 }
apply(inc)(3)         // => 4
```

`apply` can be used, together with *`core.lambda.flip()`* in higher order functions when mapping over a collection, if you want to apply them to some constant argument:

```
var fns = [
  function(a){ return a + 2 },
  function(a){ return a - 2 },
  function(a){ return a * 2 },
  function(a){ return a / 2 }
]

fns.map(flip(apply)(3)) => [5, 1, 6, 1.5]
```

Function: `flip`

`core.lambda.`**`flip`**`(f)`

Returns The function `f` with parameters inverted.

$$(\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$$

Inverts the order of the parameters of a binary function.

Examples

```
var subtract = function(a){ return function(b){ return a - b }}

subtract(3)(2)                // => 1
flip(subtract)(3)(2)          // => -1
```

Flip can be used to partially apply the second argument in a binary curried function. It makes it much easier to create new functionality, by just applying functions, rather than explicitly creating new ones:

```
var divide = curry(2, function(a, b) {
  return a / b
})

var dividedBy = curry(2, function(a, b) {
  return b / a
})

var dividedBy5 = function(a) {
  return divide(a, 5)
}

// Instead you could write:
var dividedBy  = flip(divide)
var dividedBy5 = dividedBy(5)
```

Function: `compose`

`core.lambda.`**`compose`**(f, g)

Returns A composition of `f` and `g`.

$$(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Composes two functions together.

Examples

```
function inc(a){ return a + 1 }
function square(a){ return a * a }

compose(inc)(square)(2)       // => inc(square(2)) => 5
```

Function: `curry`

`core.lambda.`**`curry`**(n, f)

Returns A curried version of `f`, up to `n` arguments.

$$\texttt{:Number} \to (\alpha_1, \alpha_2, \ldots, \alpha \to \beta) \to (\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta)$$

Transforms any function on tuples into a curried function.

Examples

```
function sub3(a, b, c){ return a - b - c }

curry(3, sub3)(5)(2)(1)       // => 2
curry(3, sub3)(5, 2)(1)       // => 2
curry(3, sub3)(5)(2, 1)       // => 2
curry(3, sub3)(5, 2, 1)       // => 2
curry(3, sub3)(5, 2, 1, 0)    // => TypeError: 2 is not a function
```

Function: spread

core.lambda.**spread**(*f*, *xs*)

Returns The result of applying the function `f` to arguments `xs`.

$(\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta) \to (\#[\alpha_1, \alpha_2, \ldots, \alpha] \to \beta)$

Applies a list of arguments to a curried function.

Examples

```
var add = curry(2, function(a, b){ return a + b })

spread(add)([3, 2])  // => add(3)(2) => 5
```

Function: uncurry

core.lambda.**uncurry**(*f*)

Returns A function on tuples.

$(\alpha_1 \to \alpha_2 \to \ldots \to \alpha \to \beta) \to (\alpha_1, \alpha_2, \ldots, \alpha \to \beta)$

Transforms a curried function into a function on tuples.

Examples

```
var add = function(a){ return function(b){ return a + b }}

uncurry(add)(3, 2)   // => add(3)(2) => 5
```

Function: upon

core.lambda.**upon**(*f*, *g*)

Returns A binary function `f` with arguments transformed by `g`.

$$(\beta \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \alpha \to \gamma)$$

Applies an unary function to both arguments of a binary function.

Examples

```
// Sorting an array of pairs by the first component
var curry = require('core.lambda').curry

var xss = [[1, 2], [3, 1], [-2, 4]]

function compare(a, b) {
  return a < b?     -1
  :      a === b?    0
  :      /* a> b */  1
}

function first(xs) {
  return xs[0]
}

function sortBy(f, xs) {
  return xs.slice().sort(f)
}

var compareC = curry(2, compare)

sortBy(upon(compareC, first), xss)  // => [[-2, 4], [1, 2], [3, 1]]
```

Module: `core.operators`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/core.operators/issues

Version 1.0.0

Repository https://github.com/folktale/core.operators

Portability Portable

npm package `core.operators`

Provides JS operators as curried functions.

Loading

Require the `core.operators` package, after installing it:

```
var operators = require('core.operators')
```

Why?

JavaScript's operators are not first-class functions, so using them in a higher-order function requires one to wrap the call at the call-site:

```
var people = [
  { name: 'Bob', age: 14 },
  { name: 'Alice', age: 12 }
]

people.map(function(person){ return person.name })
// => ['Bob', 'Alice']
```

This defeats some of the compositional nature of functional programming. This module provides first-class, curried versions of these special operators that you can combine with the usual function composition operations:

```
var op = require('core.operators')
people.map(op.get('name'))
// => ['Bob', 'Alice']

function compare(a, b) {
  return a > b?    1
  :      a === b?  0
  :   /* a < b */ -1
}

var lambda = require('core.lambda')
people.sort(lambda.upon(compare, op.get('age'))).map(op.get('name'))
// => ['Alice', 'Bob']
```

Arithmetic

add()

`core.operators.`**`add`**`(a, b)`

```
Number → Number → Number
```

JavaScript's addition (`a + b`) operator.

subtract()

`core.operators.`**`subtract`**`(a, b)`

```
Number → Number → Number
```

JavaScript's subtraction (`a - b`) operator.

divide()

core.operators.**divide**(*a*, *b*)

```
Number → Number → Number
```

JavaScript's division (`a / b`) operator.

multiply()

core.operators.**multiply**(*a*, *b*)

```
Number → Number → Number
```

JavaScript's multiplication (`a * b`) operator.

modulus()

core.operators.**modulus**(*a*, *b*)

```
Number → Number → Number
```

JavaScript's modulus (`a % b`) operator.

negate()

core.operators.**negate**(*a*)

```
Number → Number
```

JavaScript's unary negation (`-a`) operator.

increment()

core.operators.**increment**(*a*)

```
Number → Number
```

Short for `add(1)(a)`.

decrement()

core.operators.**decrement**(*a*)

```
Number → Number
```

Short for `subtract(a)(1)`.

Bitwise

bitNot()

`core.operators.`**`bitNot`**(*a*)

```
Int → Int
```

Bitwise negation (`~a`)

bitAnd()

`core.operators.`**`bitAnd`**(*a*, *b*)

```
Int → Int → Int
```

Bitwise intersection (`a & b`)

bitOr()

`core.operators.`**`bitOr`**(*a*, *b*)

```
Int → Int → Int
```

Bitwise union (`a | b`)

bitXor()

`core.operators.`**`bitXor`**(*a*, *b*)

```
Int → Int → Int
```

Bitwise exclusive union (`a ^ b`)

bitShiftLeft()

`core.operators.`**`bitShiftLeft`**(*a*, *b*)

```
Int → Int → Int
```

Bitwise left shift (`a << b`)

bitShiftRight()

core.operators.**bitShiftRight**(*a*, *b*)

```
Int → Int → Int
```

Sign-propagating bitwise right shift (`a >> b`)

bitUnsignedShiftRight()

core.operators.**bitUnsignedShiftRight**(*a*, *b*)

```
Int → Int → Int
```

Zero-fill bitwise right shift (`a >>> b`)

Logical

not()

core.operators.**not**(*a*)

```
Boolean → Boolean
```

Logical negation (`!a`).

and()

core.operators.**and**(*a*, *b*)

```
Boolean → Boolean → Boolean
```

Logical conjunction (`a && b`).

or()

core.operators.**or**(*a*, *b*)

```
Boolean → Boolean → Boolean
```

Logical disjunction (`a || b`).

Relational

equal()

`core.operators.`**`equal`**(*a*, *b*)

$\alpha \rightarrow \alpha \rightarrow$ Boolean

Strict reference equality (`a === b`).

notEqual()

`core.operators.`**`notEqual`**(*a*, *b*)

$\alpha \rightarrow \alpha \rightarrow$ Boolean

Strict reference inequality (`a !== b`).

greaterThan()

`core.operators.`**`greaterThan`**(*a*, *b*)

$\alpha \rightarrow \alpha \rightarrow$ Boolean

Greater than (`a > b`).

greaterThanOrEqualTo()

`core.operators.`**`greaterThanOrEqualTo`**(*a*, *b*)

$\alpha \rightarrow \alpha \rightarrow$ Boolean

Greater than or equal to (`a >= b`).

lessThan()

`core.operators.`**`lessThan`**(*a*, *b*)

$\alpha \rightarrow \alpha \rightarrow$ Boolean

Less than (`a < b`).

lessThanOrEqualTo()

`core.operators.`**`lessThanOrEqualTo`**(*a*, *b*)

```
α → α → Boolean
```

Less than or equal to (`a <= b`).

Special

get()

`core.operators.`**`get`**(*key*, *object*)

```
String → Object → α | Undefined
```

Property accessor (`object[key]`).

has()

`core.operators.`**`has`**(*key*, *object*)

```
String → Object → Boolean
```

Tests the existence of a property in an object (`key in object`).

isInstance()

`core.operators.`**`isInstance`**(*constructor*, *a*)

```
Function → Object → Boolean
```

Instance check (`a instanceof constructor`).

create()

`core.operators.`**`create`**(*constructor*, *...args*)

```
(new(α₁, α₂, ..., α) → β) → (α₁, α₂, ..., α) → β)
```

Constructs new objects (`new constructor(...args)`)

typeOf()

`core.operators.`**`typeOf`**`(`*a*`)`

$\alpha \rightarrow$ `String`

Returns the internal type of the object (`typeof a`)

classOf()

`core.operators.`**`classOf`**`(`*a*`)`

$\alpha \rightarrow$ `String`

Returns the internal `[[Class]]` of the object.

3.3.2 Control

Provides operations for control-flow.

- ***control.monads*** Common monadic combinators and sequencing operations.
- ***control.async*** Common operations for asynchronous control-flow with *Data.Task*.

Module: `control.monads`

Stability 1 - Experimental

Bug Tracker https://github.com/folktale/control.monads/issues

Version 0.6.0

Repository https://github.com/folktale/control.monads

Portability Portable

npm package `control.monads`

Common monadic combinators and sequencing operations.

Loading

Require the `control.monads` package, after installing it:

```
var monads = require('control.monads')
```

Uncategorised

sequence()

`control.monads.`**`sequence`**`(`*type*, *monads*`)`

Returns A monad containing an array of the values.

```
m:Monad(_) => m → Array(m(α)) → m(Array(α))
```

mapM()

control.monads.**mapM**(*type*, *transformation*, *values*)

Returns A monad containing an array of the values.

```
m:Monad(_) => m → (α → m(β)) → Array(α) → m(Array(β))
```

Converts each value into a monadic action, then evaluates such actions, left to right, and collects their results.

compose()

control.monads.**compose**(*f*, *g*, *value*)

Returns A composition of the given functions on monads.

```
m:Monad(_) => (α → m(β)) → (β → m(γ)) → α → m(γ)
```

Left-to-right Kleisi composition of monads.

rightCompose()

control.monads.**rightCompose**(*f*, *g*, *value*)

Returns A composition of the given functions on monads.

```
m:Monad(_) => (β → m(γ)) → (α → m(β)) → α → m(γ)
```

Right-to-left Kleisi composition of monads.

join()

control.monads.**join**(*monad*)

Returns The nested monad.

```
m:Monad(_) => m(m(α)) → m(α)
```

Removes one level of nesting for a nested monad.

filterM()

control.monads.**filterM**(*type*, *predicate*, *values*)

Returns An array with values that pass the predicate, inside a monad.

```
m:Monad(_) => m → (α → m(Boolean)) → Array(α) → m(Array(α))
```

Filters the contents of an array with a predicate returning a monad.

liftM2()

control.monads.**liftM2**(*transformation*, *monad1*, *monad2*)

Returns The transformed value inside a monad.

```
m:Monad(_) => (α, β → γ) → m(α) → m(β) → m(γ)
```

Promotes a regular binary function to a function over monads.

liftMN()

control.monads.**liftMN**(*transformation*, *values*)

Returns The transformed value inside a monad.

```
m:Monad(_) => (α₁, α₂, ..., α → β)
            → Array(m(α₁), m(α₂), ..., m(α))
            → m(β) :: throws
```

Promotes a regular function of arity `N` to a function over monads.

Curried methods

concat()

control.monads.**concat**(*left*, *right*)

Returns A new semigroup with the values combined.

```
s:Semigroup(_) => s(α) → s(α) → s(α)
```

Concatenates two semigroups.

empty()

control.monads.**empty**()

Returns A new empty semigroup.

```
s:Semigroup(_) => s → s(α)
```

map()

control.monads.**map**(*transformation*, *functor*)

Returns A functor with its contents transformed by `f`.

```
f:Functor(_) => (α → β) → f(α) → f(β)
```

Maps over a functor instance.

of()

`control.monads.`**`of`**(*value*, *type*)

Returns A new applicative instance containing the given value.

```
f:Applicative(_) => α → f → f(α)
```

Constructs a new applicative intance.

ap()

`control.monads.`**`ap`**(*transformation*, *applicative*)

Returns A new applicative with values transformed by the receiver.

```
f:Applicative(_) => f(α → β) → f(α) → f(β)
```

Applies the function of an Applicative to the values of another Applicative.

chain()

`control.monads.`**`chain`**(*transformation*, *monad*)

Returns A new monad as transformed by the function.

```
c:Chain(_) => (α → c(β)) → c(α) → c(β)
```

Transforms the values of a monad into a new monad.

Module: `control.async`

Stability 1 - Experimental

Bug Tracker https://github.com/folktale/control.async

Version 0.5.1

Repository https://github.com/folktale/control.async

Portability Portable

npm package `control.async`

```
Task(_, _) → AsyncModule
```

Operations for asynchronous control flow.

Loading

Require the `control.async` package, after installing it, and give it a valid *`data.task.Task`* object to instantiate it:

```
var Task = require('data.task')
var Async = require('control.async')(Task)
```

Combining tasks

parallel()

`control.async.`**`parallel`**(*tasks*)

Returns A task that runs the given ones in parallel.

```
Array(Task(α, β)) → Task(α, Array(β))
```

Resolves all tasks in parallel, and collects their results.

nondeterministicChoice()

`control.async.`**`nondeterministicChoice`**(*tasks*)

Returns A task that selects the first task to resolve.

```
Array(Task(α, β)) → Task(α, Maybe(β))
```

Runs all tasks in parallel, selects the first one to either succeed or fail.

choice()

`control.async.`**`choice`**(*tasks*)

```
Array(Task(α, β)) → Task(α, Maybe(β))
```

Alias for *nondeterministicChoice()*

tryAll()

`control.async.`**`tryAll`**(*tasks*)

```
Array(Task(α, β)) → Task(Array(α), Maybe(β))
```

Creates a task that succeeds if one task succeeds, or fails if all of them fail.

Converting

lift()

`control.async.`**`lift`**(*function*)

```
(α₁, α₂, ..., α, (β → Unit)) → (α₁, α₂, ..., α → Task(Unit, β))
```

Converts a function that takes a simple continuation to a Task.

liftNode()

`control.async.`**`liftNode`**(*function*)

```
(α₁, α₂, ..., α, (β, γ → Unit)) → (α₁, α₂, ..., α → Task(β, γ))
```

Converts a function that takes a Node-style continuation to a Task.

toNode()

`control.async.`**`toNode`**(*task*)

```
Task(α, β) → (α | null, β | null → Unit)
```

Converts a Task to a Node-style function.

fromPromise()

`control.async.`**`fromPromise`**(*promise*)

```
Promise(α, β) → Task(α, β)
```

Converts a Promises/A+ to a Task.

toPromise()

`control.async.`**`toPromise`**(*constructor*, *task*)

```
PromiseConstructor → Task(α, β) → Promise(α, β)

type PromiseConstructor = new((α → Unit), (β → Unit) → Unit)
                        → Promise(α, β)
```

Converts from Task to Promises/A+.

Note: Do note that nested Tasks, unlike Promises/A+, are **NOT** flattened. You need to manually call `control.monads.join()` until you get to the value itself, if you care about passing just the value.

Error handling

catchOnly()

`control.async.`**`catchOnly`**(*filter*, *task*)

```
(γ → Boolean) → Task(α, β) :: throws(γ) → Task(α | γ, β)
```

Reifies some errors thrown by the computation to a rejected task.

+

catchAllPossibleErrors()

`control.async.`**`catchAllPossibleErrors`**`(`*task*`)`

```
Task(α, β) :: throws(Any) → Task(Any, β)
```

Reifies **all** errors thrown by the computation to a rejected task.

+

Timers

delay()

`control.async.`**`delay`**`(`*milliseconds*`)`

Returns A Task that succeeds after N milliseconds.

```
Number → Task(Unit, Number)
```

Constructs a Task that always succeeds after at least N milliseconds. The value of the Task will be the delta from the time of its initial execution to the time it gets resolved.

timeout()

`control.async.`**`timeout`**`(`*milliseconds*`)`

Returns A Task that always fails after N milliseconds.

```
Number → Task(TimeoutError, Unit)
```

Constructs a Task that always fails after at least N milliseconds.

Transforming

memoise()

`control.async.`**`memoise`**`(`*task*`)`

```
Task(α, β) → Task(α, β)
```

Caches the result of a Task, to avoid running the same task again for idempotent or pure tasks.

catchOnly

control.async.**catchOnly**(*filter*, *task*)

```
(γ → Boolean) → Task(α, β) :: throws(γ) → Task(α | γ, β)
```

Reifies some errors thrown by the computation to a rejected task.

Ideally you wouldn't care about reifying errors thrown by synchronous computations, but this might come in handy for some lifted computations.

catchAllPossibleErrors

control.async.**catchAllPossibleErrors**(*task*)

```
Task(α, β) :: throws(Any) → Task(Any, β)
```

Reifies **all** errors thrown by the computation to a rejected task.

Ideally you wouldn't care about reifying errors thrown by synchronous computations, but this might come in handy for some lifted computations.

> **Warning: Special care should be taken when using this method, since it'll reify *ALL* errors (for example, OutOfMemory errors, StackOverflow errors, ...), and it can potentially lead the whole system to an unstable state.** The `catchOnly()` function is favoured over this one, since you can decide which errors should be caught and reified in the task, and have all the others crash the process as expected.

3.3.3 Data

Provides functional (persistent and immutable) data structures for representing program data.

- ***data.either*** Right-biased disjunctions. Commonly used for modelling computations that may fail with additional information about the failure.
- ***data.maybe*** Safe optional values. Commonly used for modelling computations that may fail, or values that might not be available.
- ***data.task*** A structure for capturing the effects of time-dependent values (asynchronous computations, latency, etc.) with automatic resource management.
- ***data.validation*** A disjunction for validating inputs and aggregating failures. Isomorphic to *Data.Either*.

Module: data.either

Stability 3 - Stable

Bug Tracker https://github.com/folktale/data.either/issues

Version 1.2.0

Repository https://github.com/folktale/data.either

Portability Portable

npm package `data.either`

A structure for disjunctions (e.g.: computations that may fail).

The `Either(α, β)` structure represents the logical disjunction between α and β. In other words, `Either` may contain either a value of type α, or a value of type β, at any given time, and it's possible to know which kind of value is contained in it.

This particular implementation is biased towards right values (β), thus common projections (e.g.: for the monadic instance) will take the right value over the left one.

Loading

Require the `data.either` package, after installing it:

```
var Either = require('data.either')
```

This gives you back an *data.either.Either* object.

Why?

A common use of this structure is to represent computations that may fail when you want to provide additional information on the failure. This can force failures and their handling to be explicit, avoiding the problems associated with throwing exceptions: non locality, abnormal exits, unintended stack unwinding, etc.

Additional resources

- A Monad in Practicality: First-Class Failures — A tutorial showing how the Either data structure can be used to model failures.

Types and structures

Either

class `data.either.`**Either**

```
type Either(α, β) = Left(α) | Right(β)

implements
  Applicative(β), Functor(β), Chain(β), Monad(β), ToString
```

Represents the logical disjunction between α and β.

+

Type: `Either`

class `data.either.`**Either**

```
type Either(α, β) = Left(α) | Right(β)

implements
  Applicative(β), Functor(β), Chain(β), Monad(β), ToString
```

Represents the logical disjunction between α and β.

Comparing and testing

#isLeft

`Either.prototype.`**`isLeft`**

```
Boolean
```

True if the `Either(α, β)` contains a `Left` value.

#isRight

`Either.prototype.`**`isRight`**

```
Boolean
```

True if the `Either(α, β)` contains a `Right` value.

#isEqual()

`Either.prototype.`**`isEqual`**`(`*anEither*`)`

```
@Either(α, β) => Either(α, β) → Boolean
```

Tests if two `Either(α, β)` structures are equal. Compares the contents using reference equality.

Constructing

.Left()

static `Either.`**`Left`**`(`*value*`)`

```
α → Either(α, β)
```

Constructs a new `Either(α, β)` structure holding a `Left` value. This usually represents a failure, due to the right-bias of this structure.

.Right()

static `Either.`**`Right`**`(`*value*`)`

```
β → Either(α, β)
```

Constructs a new `Either(α, β)` structure holding a `Right` value. This usually represents a successful value due to the right bias of this structure.

.of()

static `Either.`**`of`**`(`*value*`)`

```
β → Either(α, β)
```

Creates a new `Either(α, β)` instance holding the `Right` value β.

.fromNullable()

static `Either.`**`fromNullable`**`(`*value*`)`

```
α | null | undefined → Either(null | undefined, α)
```

Constructs a new `Either(α, β)` structure from a nullable type.

Takes the `Left` value if the value is `null` or `undefined`. Takes the `Right` case otherwise.

.fromValidation()

static `Either.`**`fromValidation`**`(`*value*`)`

```
Validation(α, β) → Either(α, β)
```

Constructs a new `Either(α, β)` structure from a `Validation(α, β)` structure.

Converting

#toString()

`Either.prototype.`**`toString`**`()`

```
@Either(α, β) => Unit → String
```

Returns a textual representation of the `Either(α, β)` structure.

Extracting

#get()

`Either.prototype.`**`get`**`()`

Raises

- **TypeError** - If the structure has no `Right` value.

```
@Either(α, β) => Unit → β :: throws
```

Extracts the `Right` value out of the `Either(α, β)` structure, if it exists.

#getOrElse()

`Either.prototype.`**`getOrElse`**`(`*`default`*`)`

```
@Either(α, β) => β → β
```

Extracts the `Right` value out of the `Either(α, β)` structure. If it doesn't exist, returns a default value.

#merge()

`Either.prototype.`**`merge`**`()`

```
@Either(α, β) => Unit → α | β
```

Returns whichever side of the disjunction that is present.

Transforming

#ap()

`Either.prototype.`**`ap`**`(`*`anApplicative`*`)`

```
@Either(α, β → γ), f:Applicative(_) => f(β) → f(γ)
```

Applies the function inside the `Either(α, β)` structure to another Applicative type.

#map()

`Either.prototype.`**`map`**`(`*`transformation`*`)`

```
@Either(α, β) => (β → γ) → Either(α, γ)
```

Transforms the `Right` value of the `Either(α, β)` structure using a regular unary function.

#chain()

`Either.prototype.`**`chain`**(*transformation*)

```
@Either(α, β), m:Monad(_) => (β → m(γ)) → m(γ)
```

Transforms the `Right` value of the `Either(α, β)` structure using an unary function over monads.

#fold()

`Either.prototype.`**`fold`**(*leftTransformation*, *rightTransformation*)

```
@Either(α, β) => (α → γ), (β → γ) → γ
```

Applies a function to each case in the data structure.

#cata()

`Either.prototype.`**`cata`**(*pattern*)

```
@Either(α, β) => { r | Pattern } → γ
type Pattern {
  Left: α → γ,
  Right: β → γ
}
```

Applies a function to each case in the data structure.

#swap()

`Either.prototype.`**`swap`**()

```
@Either(α, β) => Unit → Either(β, α)
```

Swaps the disjunction values.

#bimap()

`Either.prototype.`**`bimap`**(*leftTransformation*, *rightTransformation*)

```
@Either(α, β) => (α → γ), (β → δ) → Either(γ, δ)
```

Maps both sides of the disjunction.

#leftMap()

`Either.prototype.`**`leftMap`**`(`*transformation*`)`

```
@Either(α, β) => (α → γ) → Either(γ, β)
```

Maps the left side of the disjunction.

#orElse()

`Either.prototype.`**`orElse`**`(`*transformation*`)`

```
@Either(α, β) => (α → Either(γ, β)) → Either(γ, β)
```

Transforms the `Left` value into a new `Either(α, β)` structure.

Module: `data.maybe`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/data.maybe/issues

Version 1.2.0

Repository https://github.com/folktale/data.maybe

Portability Portable

npm package `data.maybe`

A structure for values that may not be present, or computations that may fail.

The class models two different cases:

- `Just α` — represents a `Maybe(α)` that contains a value. `α` may be any value, including `null` and `undefined`.
- `Nothing` — represents a `Maybe(α)` that has no values. Or a failure that needs no additional information.

Loading

Require the `data.maybe` package, after installing it:

```
var Maybe = require('data.maybe')
```

This gives you back a *`data.maybe.Maybe`* object.

Why?

The `Maybe(α)` structure explicitly models the effects that are implicit in `Nullable` types, thus has none of the problems associated with using `null` or `undefined`, such as `NullPointerException` or `TypeError: undefined is not a function`.

Common uses of this structure includes modelling values that may or may not be present. For example, instead of having both a `collection.has(a)` and `collection.get(a)` operation, one may have the `collection.get(a)` operation return a `Maybe(α)` value. This avoids a problem of data incoherence (specially in asynchronous collections, where a value may be added between a call to `.has()` and `.get()`!).

Another common usage is for modelling functions that might fail to provide a value. E.g.: `collection.find(predicate)` can safely return a `Maybe(α)` instance, even if the collection allows nullable values.

Additional resources

- A Monad in Practicality: First-Class Failures — A tutorial showing how the Either data structure can be used to model failures.

Types and structures

Maybe

class `data.maybe.`**`Maybe`**

```
type Maybe(α) = Nothing | Just(α)

implements
  Applicative(α), Functor(α), Chain(α), Monad(α), ToString
```

A structure for values that may not be present, or computations that may fail.

+

Type: `Maybe`

class `data.maybe.`**`Maybe`**

```
type Maybe(α) = Nothing | Just(α)

implements
  Applicative(α), Functor(α), Chain(α), Monad(α), ToString
```

A structure for values that may not be present, or computations that may fail.

Comparing and testing

#isNothing

`Maybe.prototype.`**`isNothing`**

```
Boolean
```

True if the `Maybe(α)` structure contains a `Nothing`.

#isJust

`Maybe.prototype.`**`isJust`**

```
Boolean
```

True if the `Maybe(α)` structure contains a `Just`.

#isEqual()

`Maybe.prototype.`**`isEqual`**`(`*aMaybe*`)`

```
@Maybe(α) => Maybe(α) → Boolean
```

Tests if two `Maybe(α)` contains are similar.

Contents are checked using reference equality.

Constructing

.Nothing()

static `Maybe.`**`Nothing`**`()`

```
Unit → Maybe(α)
```

Constructs a new `Maybe(α)` structure with an absent value. Commonly used to represent a failure.

.Just()

static `Maybe.`**`Just`**`(`*value*`)`

```
α → Maybe(α)
```

Constructs a new `Maybe(α)` structure that holds the single value α. Commonly used to represent a success.

.of()

static `Maybe.`**`of`**`(`*value*`)`

```
α → Maybe(α)
```

Constructs a new `Maybe(α)` structure that holds the single value α.

.fromNullable()

static `Maybe.`**`fromNullable`**`(`*value*`)`

```
α | null | undefined → Maybe(α)
```

Constructs a new `Maybe(α)` value from a nullable type.

If the value is `null` or `undefined`, returns a `Nothing`, otherwise returns the value wrapped in a `Just`.

.fromEither()

static `Maybe.`**`fromEither`**`(`*value*`)`

```
Either(α, β) → Maybe(β)
```

Constructs a new `Maybe(β)` from an `Either(α, β)` value.

.fromValidation()

static `Maybe.`**`fromValidation`**`(`*value*`)`

```
Validation(α, β) → Maybe(β)
```

Constructs a new `Maybe(β)` from a `Validation(α, β)` value.

Converting

#toString()

`Maybe.prototype.`**`toString`**`()`

```
@Maybe(α) => Unit → String
```

Returns a textual representation of the structure.

#toJSON()

`Maybe.prototype.`**`toJSON`**`()`

```
@Maybe(α) => Unit → Object
```

Returns a JSON serialisation of the structure.

Extracting

#get()

`Maybe.prototype.`**`get`**`()`

Raises

- **TypeError** - if the structure is a Nothing.

```
@Maybe(α) => Unit → α :: throws
```

Extracts the value out of the structure, if it exists.

#getOrElse()

`Maybe.prototype.`**`getOrElse`**`(`*`default`*`)`

```
@Maybe(α) => α → α
```

Extracts the value out of the structure, if it exists. Otherwise return the given default value.

Transforming

#ap()

`Maybe.prototype.`**`ap`**`(`*`anApplicative`*`)`

```
@Maybe(α → β), f:Applicative(_) => f(α) → f(β)
```

Applies the function inside the structure to another Applicative type.

#map()

`Maybe.prototype.`**`map`**`(`*`transformation`*`)`

```
@Maybe(α) => (α → β) → Maybe(β)
```

Transforms the value of this structure using a regular unary function.

#chain()

`Maybe.prototype.`**`chain`**`(`*`transformation`*`)`

```
@Maybe(α), m:Monad(_) => (α → m(β)) → m(β)
```

Transforms the value of this structure using an unary function over monads.

#orElse()

`Maybe.prototype.`**`orElse`**`(`*transformation*`)`

```
@Maybe(α) => (Unit → Maybe(β)) → Maybe(β)
```

Transforms the failure into a new `Maybe` structure.

#cata()

`Maybe.prototype.`**`cata`**`(`*aPattern*`)`

```
@Maybe(α) => { Nothing: Unit → β, Just: α → β } → β
```

Applies a function to each case in the data structure.

Module: `data.task`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/data.task/issues

Version 3.0.0

Repository https://github.com/folktale/data.task

Portability Portable

npm package `data.task`

A structure for time-dependent values, providing explicit effects for delayed computations, latency, etc.

Loading

Require the `data.task` package, after installing it:

```
var Task = require('data.task')
```

This gives you back a *`data.task.Task`* object.

Why?

This structure allows one to model side-effects (specially time-based ones) explicitly, such that one can have full knowledge of when they're dealing with delayed computations, latency, or anything that isn't pure or can be computed immediately.

A common use of this structure is to replace the usual Continuation-Passing Style form of programming in order to be able to compose and sequence time-dependent effects using the generic and powerful monadic operations.

Additional resources

- A Monad in Practicality: Controlling Time — A tutorial showing how `Data.Task` can be used to model time-dependent values.

Types and structures

Task

class `data.task.`**`Task`**

```
type Task(α, β)

new ((α → Unit), (β → Unit) → γ), (γ → Unit)

implements
  Chain(β), Monad(β), Functor(β), Applicative(β),
  Semigroup(β), Monoid(β), ToString
```

A structure for time-dependent values.

+

Type: `Task`

class `data.task.`**`Task`**

```
type Task(α, β)

new ((α → Unit), (β → Unit) → γ), (γ → Unit)

implements
  Chain(β), Monad(β), Functor(β), Applicative(β),
  Semigroup(β), Monoid(β), ToString
```

A structure for time-dependent values.

Combining

#concat()

`Task.prototype.`**`concat`**(*task*)

```
@Task(α, β) => Task(α, β) → Task(α, β)
```

Selects the earlier of two Tasks.

Constructing

.of()

static `Task.`**`of`**`(`*value*`)`

```
β → Task(α, β)
```

Constructs a new Task containing the given successful value.

.rejected()

static `Task.`**`rejected`**`(`*value*`)`

```
α → Task(α, β)
```

Constructs a new Task containing the given failure value.

.empty()

static `Task.`**`empty`**`()`

```
Unit → Task(α, β)
```

Constructs a Task that will never resolve.

Transforming

#map()

`Task.prototype.`**`map`**`(`*transformation*`)`

```
@Task(α, β) => (β → γ) → Task(α, γ)
```

Transforms the successful value of the Task using a regular unary function.

#chain()

`Task.prototype.`**`chain`**`(`*transformation*`)`

```
@Task(α, β) => (β → Task(α, γ)) → Task(α, γ)
```

Transforms the succesful value of the Task using a function over monads.

#ap()

`Task.prototype.`**`ap`**`(`*task*`)`

```
@Task(α, β → γ) => Task(α, β) → Task(α, γ)
```

Transforms a Task by applying the function inside this receiver.

#orElse()

`Task.prototype.`**`orElse`**`(`*transformation*`)`

```
@Task(α, β) => (α → Task(γ, β)) → Task(γ, β)
```

Transforms the failure value of the Task into a new Task.

#fold()

`Task.prototype.`**`fold`**`(`*onRejection*, *onSucecss*`)`

```
@Task(α, β) => (α → γ), (β → γ) → Task(δ, γ)
```

Applies a function to each side of the task.

#cata()

`Task.prototype.`**`cata`**`(`*pattern*`)`

```
@Task(α, β) => { Rejected: α → γ, Resolved: β → γ } → Task(δ, γ)
```

Applies a function to each side of the task.

#swap()

`Task.prototype.`**`swap`**`()`

```
@Task(α, β) => Unit → Task(β, α)
```

Swaps the values in the task.

#bimap()

`Task.prototype.`**`bimap`**`(`*onRejection*, *onSuccess*`)`

```
@Task(α, β) => (α → γ), (β → δ) → Task(γ, δ)
```

Maps both sides of the task.

#rejectedMap()

`Task.prototype.`**`rejectedMap`**(*transformation*)

```
@Task(α, β) => (α → γ) → Task(γ, β)
```

Maps the failure side of the task.

Module: `data.validation`

Stability 3 - Stable

Bug Tracker https://github.com/folktale/data.validation/issues

Version 1.3.0

Repository https://github.com/folktale/data.validation

Portability Portable

npm package `data.validation`

A disjunction that is more appropriate for validating inputs and aggregating failures.

Loading

Require the `data.validation` package, after installing it:

```
var Validation = require('data.validation')
```

This gives you back a *`data.validation.Validation`* object.

Why?

The `Validation(α, β)` is a disjunction that's more appropriate for validating inputs, and aggregating failures. It's isomorphic to *`data.either`*, but provides better terminology for these use cases (`Failure` and `Success`, versus `Left` and `Right`), and allows one to aggregate failures and successes as an Applicative Functor.

Additional resources

- A Monad in Practicality: First-Class Failures — A tutorial showing how the Validation data structure can be used to model data validations.

Types and structures

Validation

class `data.validation.`**`Validation`**

```
type Validation(α, β) = Failure(α) | Success(β)

implements
  Applicative(β), Functor(β), ToString
```

Represents the logical disjunction between α and β.

+

Type: `Validation`

class `data.validation.`**`Validation`**

```
type Validation(α, β) = Failure(α) | Success(β)

implements
  Applicative(β), Functor(β), ToString
```

Represents the logical disjunction between α and β.

Comparing and testing

#isFailure

`Validation.prototype.`**`isFailure`**

```
Boolean
```

True if the `Validation(α, β)` contains a `Failure` value.

#isSuccess

`Validation.prototype.`**`isSuccess`**

```
Boolean
```

True if the `Validation(α, β)` contains a `Success` value.

#isEqual()

`Validation.prototype.`**`isEqual`**`(`*aValidation*`)`

```
@Validation(α, β) => Validation(α, β) → Boolean
```

Tests if two `Validation(α, β)` structures are equal. Compares the contents using reference equality.

Constructing

.Failure()

static `Validation.`**`Failure`**`(`*value*`)`

```
α → Validation(α, β)
```

Constructs a new Validation structure holding a `Failure` value.

.Success()

static `Validation.`**`Success`**`(`*value*`)`

```
β → Validation(α, β)
```

Constructs a new Validation structure holding a `Success` value.

.of()

static `Validation.`**`of`**`(`*value*`)`

```
β → Validation(α, β)
```

Creates a new Validation instance holding the `Success` value β.

.fromNullable()

static `Validation.`**`fromNullable`**`(`*value*`)`

```
α | null | undefined → Validation(null | undefined, α)
```

Constructs a new Validation structure from a nullable type.

Takes the `Failure` value if the value is `null` or `undefined`. Takes the `Success` case otherwise.

.fromEither()

static `Validation.`**`fromEither`**`(`*value*`)`

```
Either(α, β) → Validation(α, β)
```

Constructs a new `Validation(α, β)` structure from an `Either(α, β)` structure.

Converting

#toString()

`Validation.prototype.`**`toString`**`()`

```
@Validation(α, β) => Unit → String
```

Returns a textual representation of the `Validation(α, β)` structure.

Extracting

#get()

`Validation.prototype.`**`get`**`()`

Raises

- **TypeError** - If the structure has no `Success` value.

```
@Validation(α, β) => Unit → β :: throws
```

Extracts the `Success` value out of the `Validation(α, β)` structure, if it exists.

#getOrElse()

`Validation.prototype.`**`getOrElse`**`(`*default*`)`

```
@Validation(α, β) => β → β
```

Extracts the `Success` value out of the `Validation(α, β)` structure. If it doesn't exist, returns a default value.

#merge()

`Validation.prototype.`**`merge`**`()`

```
@Validation(α, β) => Unit → α | β
```

Returns whichever side of the disjunction that is present.

Transforming

#ap()

`Validation.prototype.`**`ap`**`(`*anApplicative*`)`

```
@Validation(α, β → γ), f:Applicative(_) => f(β) → f(γ)
```

Applies the function inside the `Validation(α, β)` structure to another Applicative type, and combines failures with a semigroup.

#map()

`Validation.prototype.`**`map`**`(`*transformation*`)`

```
@Validation(α, β) => (β → γ) → Validation(α, γ)
```

Transforms the `Success` value of the `Validation(α, β)` structure using a regular unary function.

#fold()

`Validation.prototype.`**`fold`**`(`*leftTransformation*, *rightTransformation*`)`

```
@Validation(α, β) => (α → γ), (β → γ) → γ
```

Applies a function to each case in the data structure.

#cata()

`Validation.prototype.`**`cata`**`(`*pattern*`)`

```
@Validation(α, β) => { r | Pattern } → γ
type Pattern {
  Failure: α → γ,
  Success: β → γ
}
```

Applies a function to each case in the data structure.

#swap()

`Validation.prototype.`**`swap`**`()`

```
@Validation(α, β) => Unit → Validation(β, α)
```

Swaps the disjunction values.

#bimap()

`Validation.prototype.`**`bimap`**`(`*leftTransformation*, *rightTransformation*`)`

```
@Validation(α, β) => (α → γ), (β → δ) → Validation(γ, δ)
```

Maps both sides of the disjunction.

#failureMap()

`Validation.prototype.`**`failureMap`**`(`*transformation*`)`

```
@Validation(α, β) => (α → γ) → Validation(γ, β)
```

Maps the left side of the disjunction.

#orElse()

`Validation.prototype.`**`orElse`**`(`*transformation*`)`

```
@Validation(α, β) => (α → Validation(γ, β)) → Validation(γ, β)
```

Transforms the `Failure` value into a new `Validation(α, β)` structure.

3.4 How do I...

3.5 Glossary

Python Module Index

c

d

Index

D

E

F

G