

Marius Jurgelenas & Karolis Masiulis

Functional Programming in the Fantasy Land



VilniusJs
2018



Functional Programming in the Fantasy Land
By Marius Jurgelenas



Create a presentation like this




$$(\mathbb{X}) \Rightarrow \mathbb{X}$$





Create a presentation like this








fantasyland / **fantasy-land**

 Code




 Issues 25


 Pull requests 8

 Projects 0

 Insights

Let the fantasy begin

 master  v3.5.0  ... 1.0.0

 puffnfresh committed on Apr 12, 2013

0 parents commit 8586eb750c6fda88210fffc8302e44ce5fb7bf5e8


Watch 201

Star 5,385

Fork 220

Browse files



Create a presentation like this 

Fantasy Land

<https://github.com/fantasyland/fantasy-land>



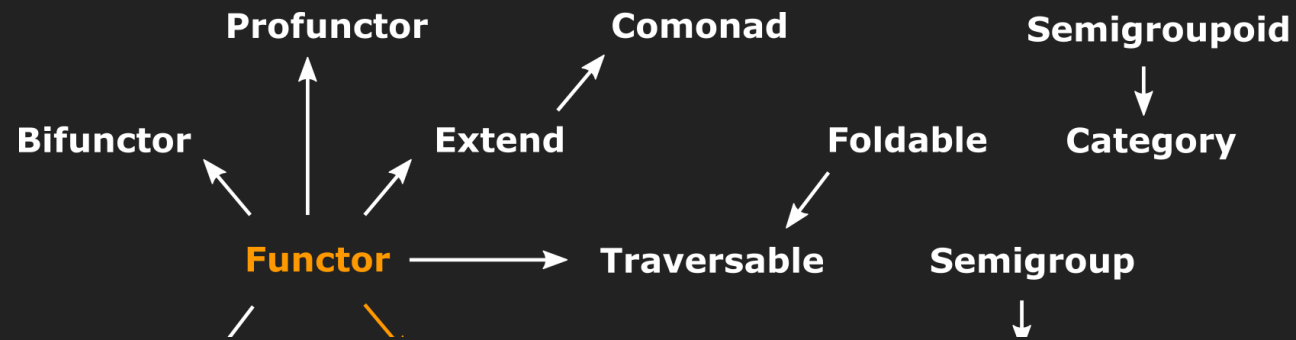
- Method Signatures
- Laws
- Constraints

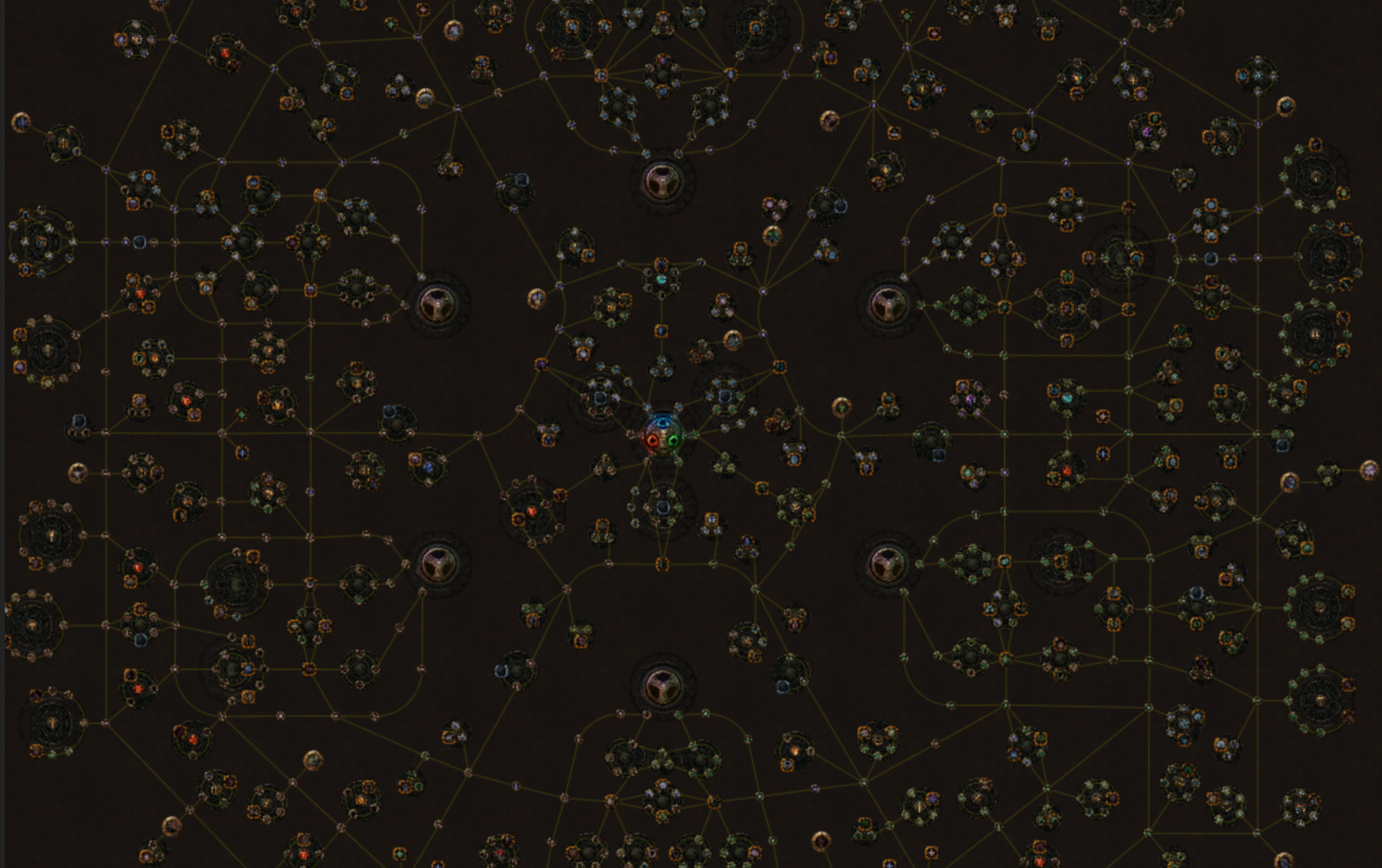


Create a presentation like this



Specified in JavaScript by Fantasy Land spec





Functor
Applicative
Apply
Chain
Monad



Create a presentation like this



Functor

Method signatures

```
map :: Functor f => f a ~> (a -> b) -> f b
```

Laws

Identity

```
u.map(a => a) === u
```

Composition

```
u.map(x => f(g(x))) === u.map(g).map(f)
```

There's a vertical slide below

Use the controls to the right or the keyboard arrows



Create a presentation like this



Extended Hindley-Milner type system

method

type

type

return

```
[1, 2, 3].map(x => x + 1) // [2, 3, 4]
```



Create a presentation like this



Functor

Constraints

```
u.map(f)
```

1. f must be a function,
 - If f is not a function, the behaviour of `map` is unspecified.
 - f can return any value.
 - No parts of f 's return value should be checked.
2. `map` must return a value of the same Functor



Create a presentation like this



Functor
Applicative
Apply
Chain
Monad



Create a presentation like this



Applicative

Method signature

```
of :: Applicative f => a -> f a
```

A value which has an Applicative must provide an **of** function on its **type representative**. The of function takes one argument:
`F.of (a)`



Create a presentation like this



Functor
Applicative
Apply
Chain
Monad



Create a presentation like this



Apply

Method signature

```
ap :: Apply f => f a ~> f (a -> b) -> f b
```



Create a presentation like this



Functor vs Apply

```
map :: Functor f => f a ~> (a -> b) -> f b
```

```
ap :: Apply f => f a ~> f (a -> b) -> f b
```



Create a presentation like this



Functor
Applicative
Apply
Chain
Monad



Create a presentation like this



Chain

Method signature

```
chain :: Chain f => f a ~> (a -> f b) -> f b
```



Create a presentation like this



Functor vs Apply vs Chain

```
map :: Functor f => f a ~> (a -> b) -> f b
```

```
ap :: Apply f => f a ~> f (a -> b) -> f b
```

```
chain :: Chain f => f a ~> (a -> f b) -> f b
```



Create a presentation like this



Functor
Applicative
Apply
Chain
Monad



Create a presentation like this



Monad

A Monad must implement the **Applicative** and **Chain** specifications.

There is one operation to **manipulate** values of the monad (**chain**), and an operation to **put** values into a monad (**of**)



Create a presentation like this



Thank you!



Create a presentation like this

