

Functional Programming beyond map/filter/reduce

JFokus 2018



Prof. Dierk König
canoo



mittie



Monads
are just
monoids in the category
of endofunctors.

Explanation Path

Monoid *it's simple*

Functor *you already know it*

Endo- *cool idea*

Monoid of endofunctors = monad

Integers (+)

$$2 + 3 == 5$$

$$2 + 0 == 2$$

$$0 + 3 == 3$$

$$(1 + 2) + 3 == 1 + (2 + 3)$$

Integers (*)

$$2 * 3 == 6$$

$$2 * 1 == 2$$

$$1 * 3 == 3$$

$$(2 * 3) * 4 == 2 * (3 * 4)$$

Boolean (&&)

`a && b` `==` `c`

`a && true` `==` `a`

`true && b` `==` `b`

`(a && b) && c` `==` `a && (b && c)`

Lists, Arrays (+)

`[1] + [2] == [1, 2]`

`[1] + [] == [1]`

`[] + [2] == [2]`

`([1]+[2])+[3] == [1]+([2]+[3])`

Strings (+)

"1" + "2" == "12"

"1" + "" == "1"

"" + "2" == "2"

("Hi" + ", ") + "JFokus" ==
"Hi" + (", " + "JFokus")

Extract

type $\langle \rangle$ type \Rightarrow type

neutral element

left & right identity

associative

Abstract

We call this concept "**Monoid**"
short for "*that thing, you know, that you can combine
with another thing and get the same result no matter which two you combine first.*"



← Als Antwort an @chris_martin

Monoid Mary @argumatronic
everything worth doing is monoidal



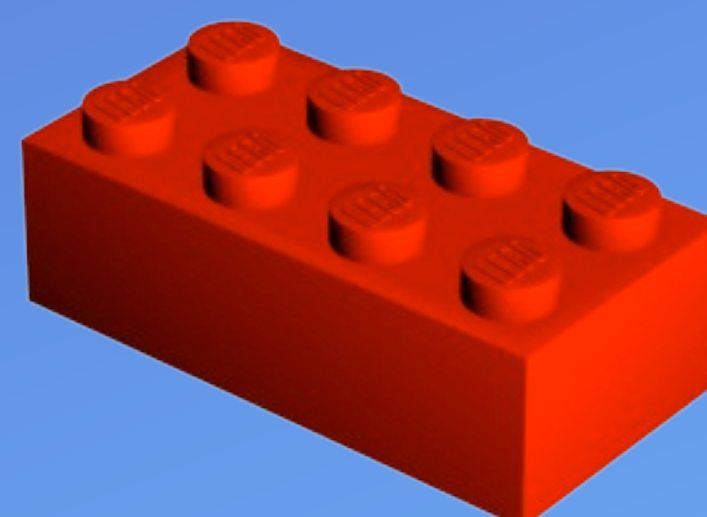
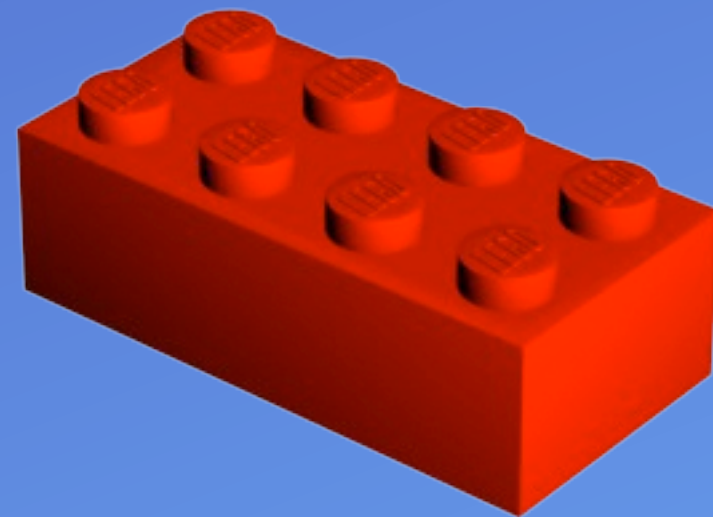
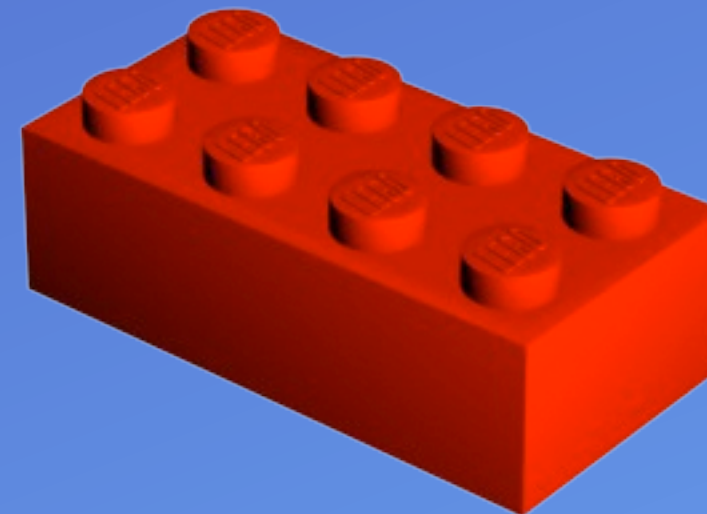
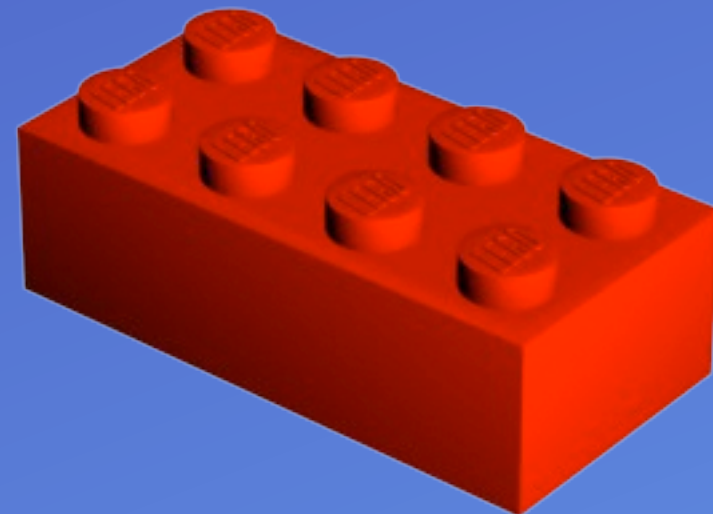
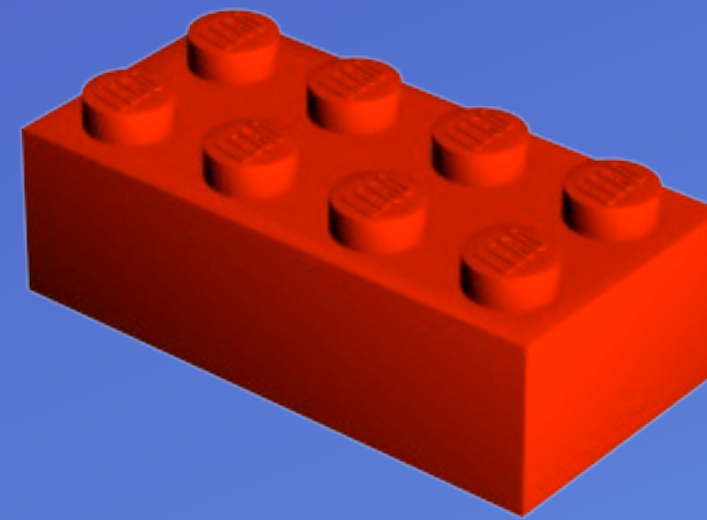
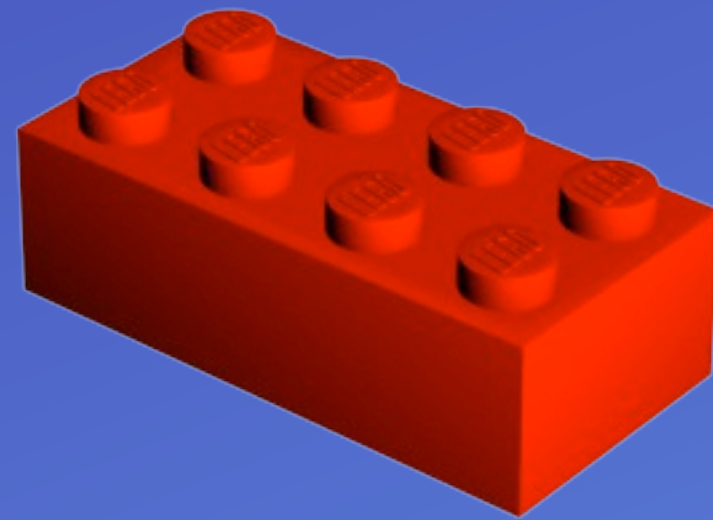
2



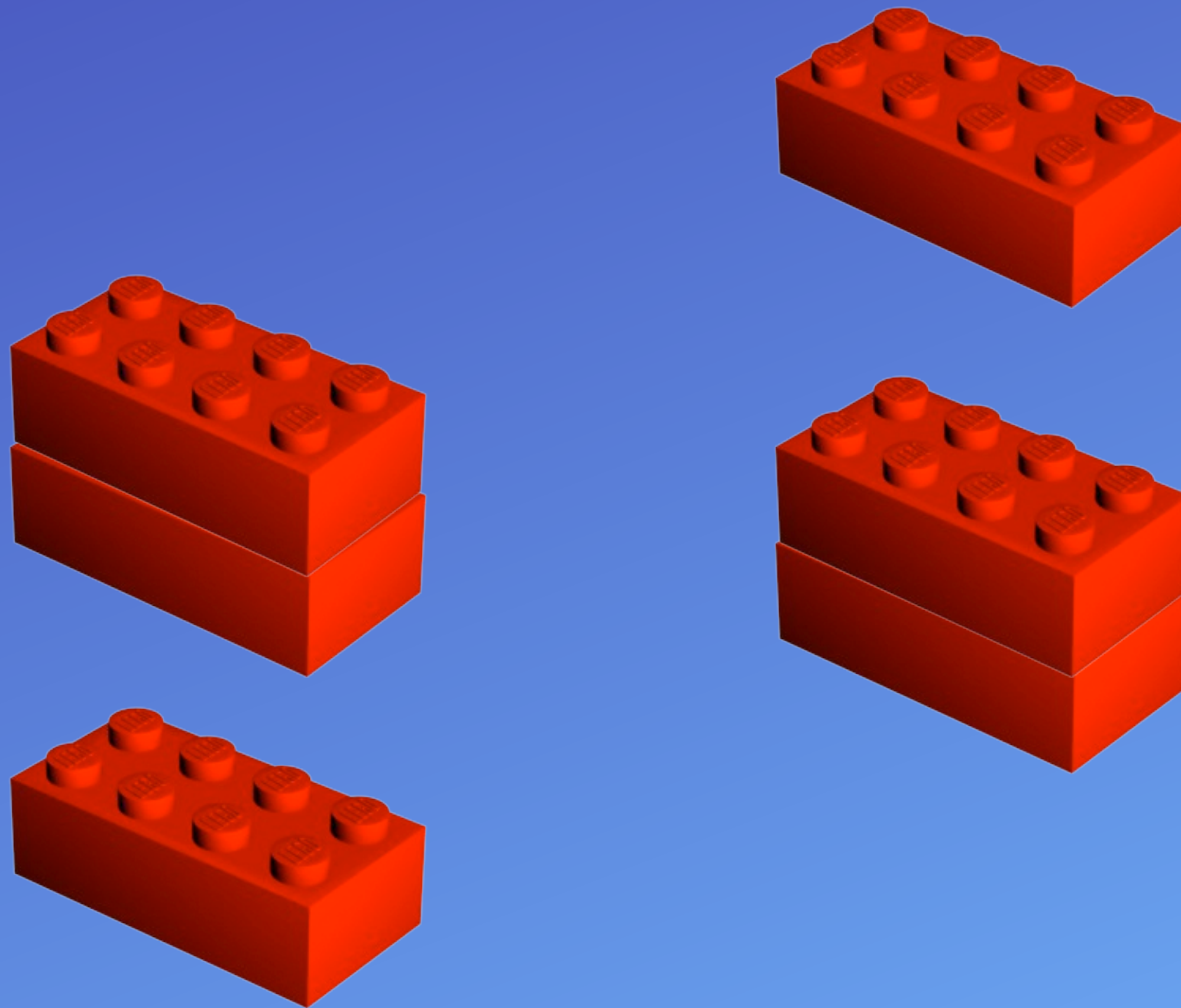
27

1d

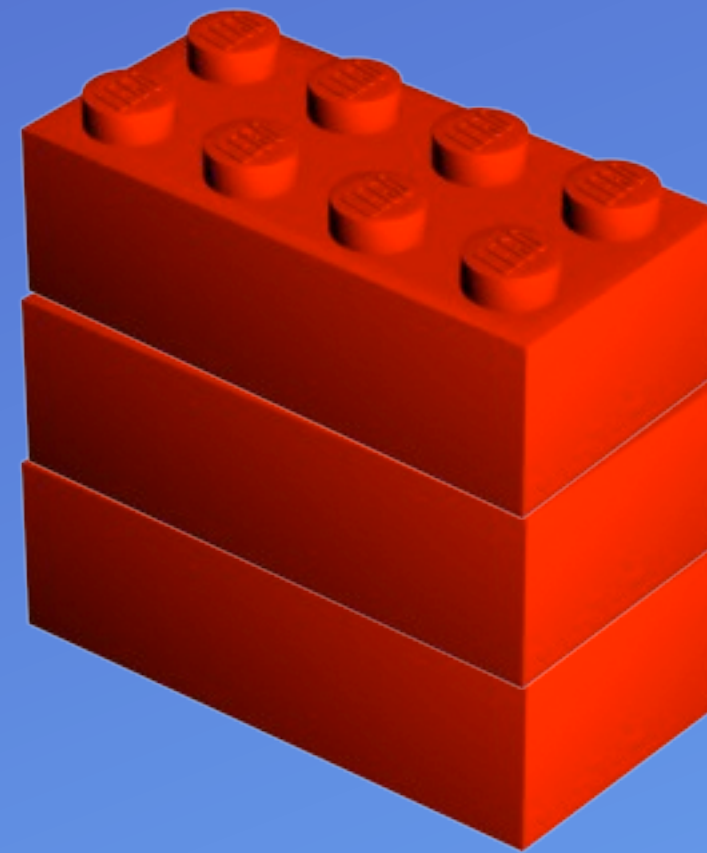
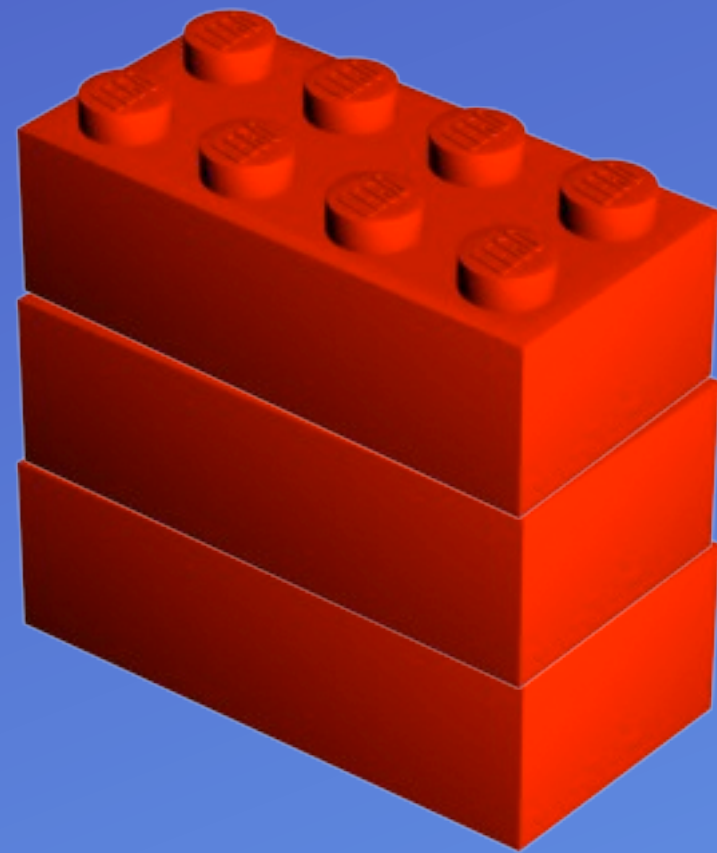
LEGO Monoid



LEGO Monoid

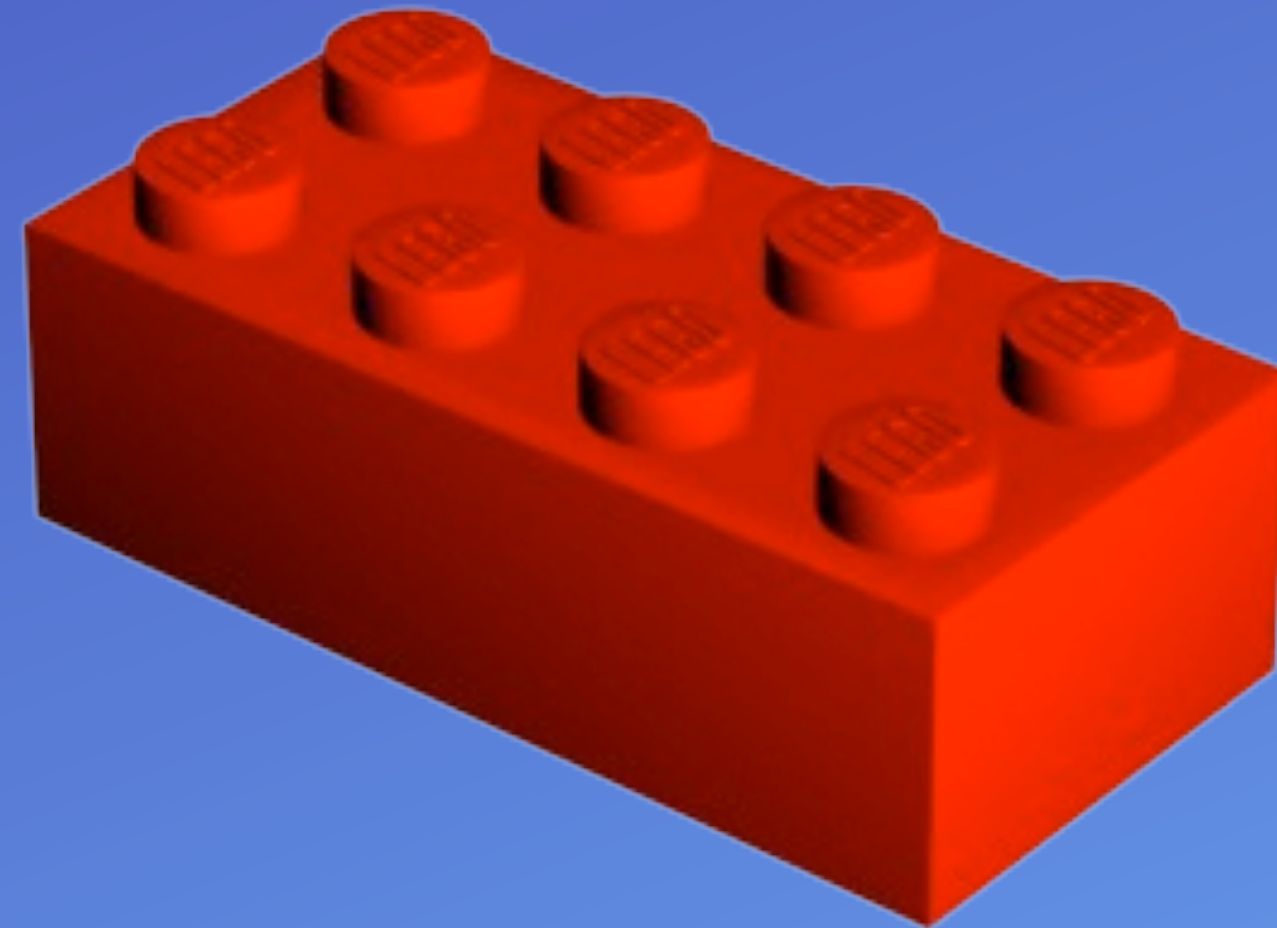


LEGO Monoid

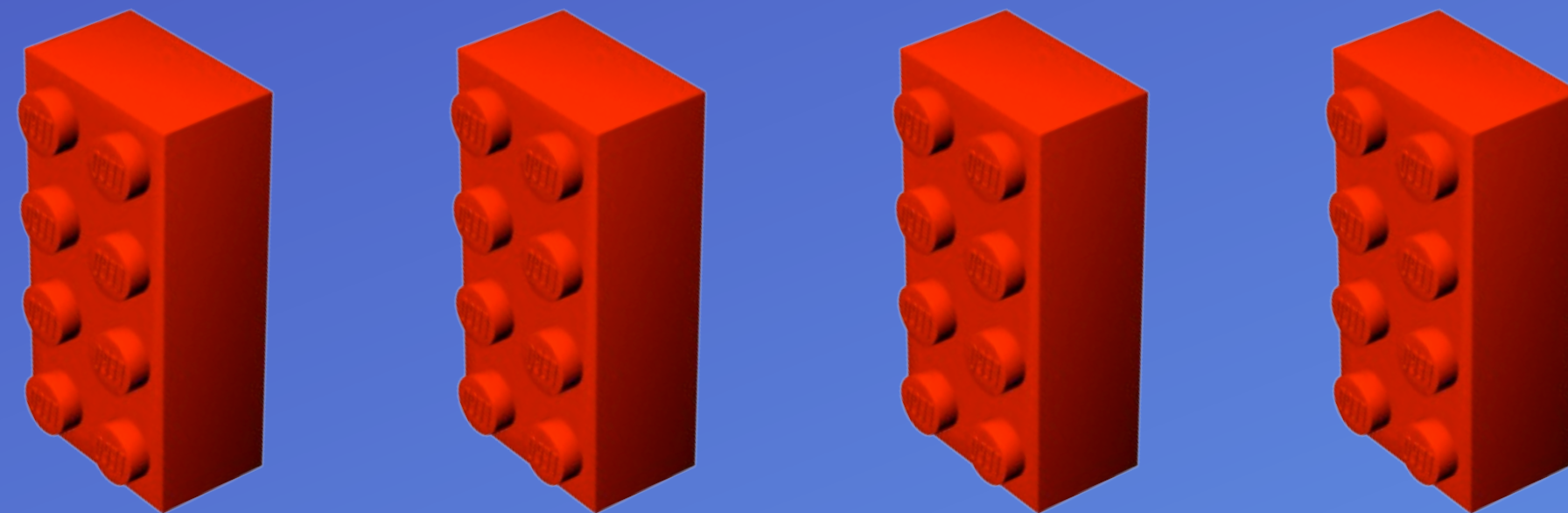


LEGO Neutral Brick

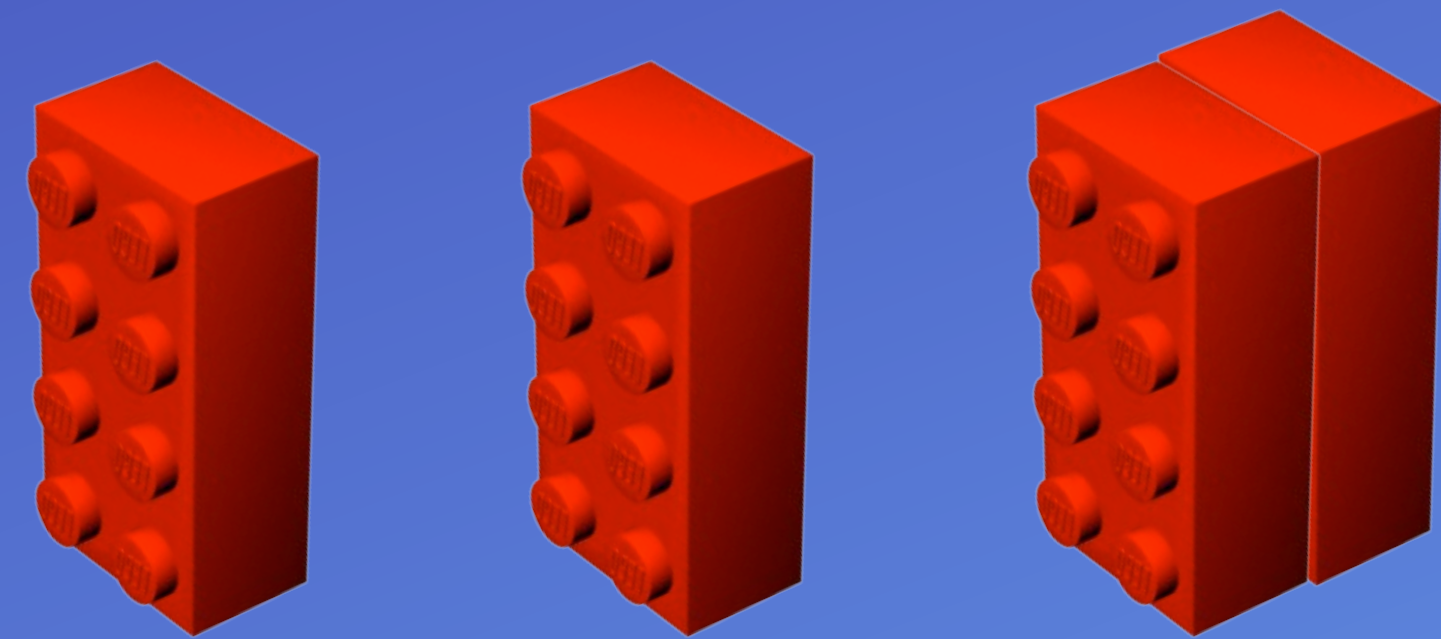
LEGO Left & Right Identity



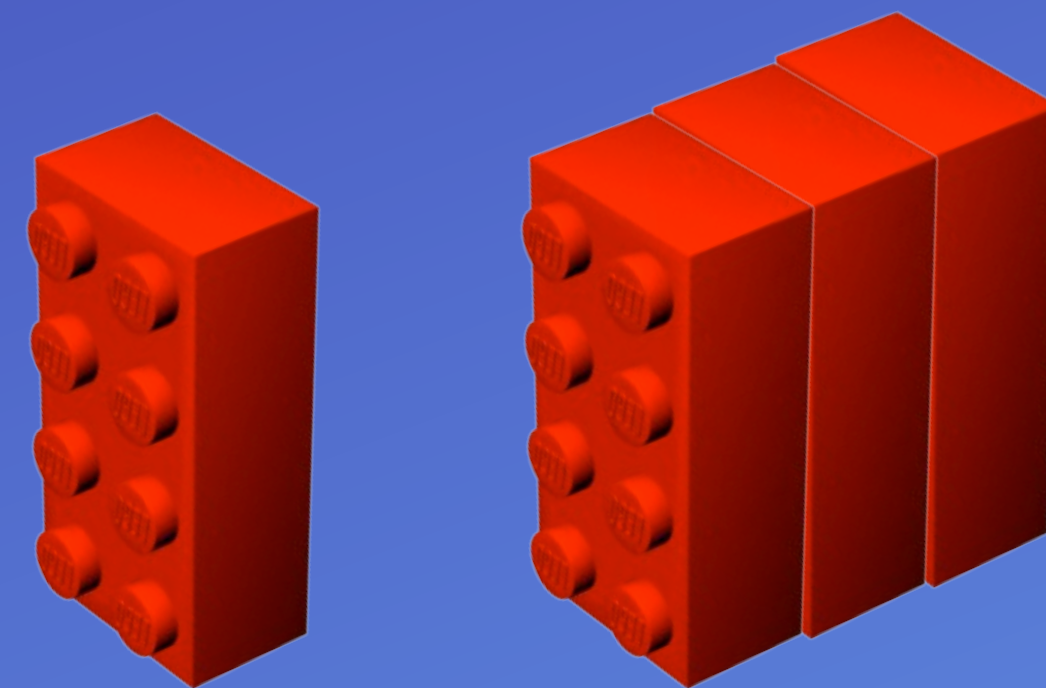
Generic Fold Right



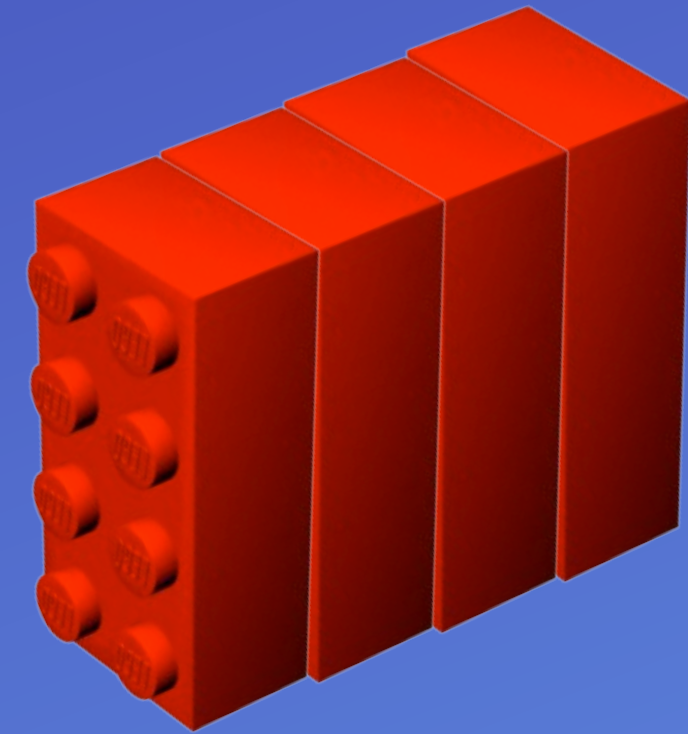
Generic Fold Right



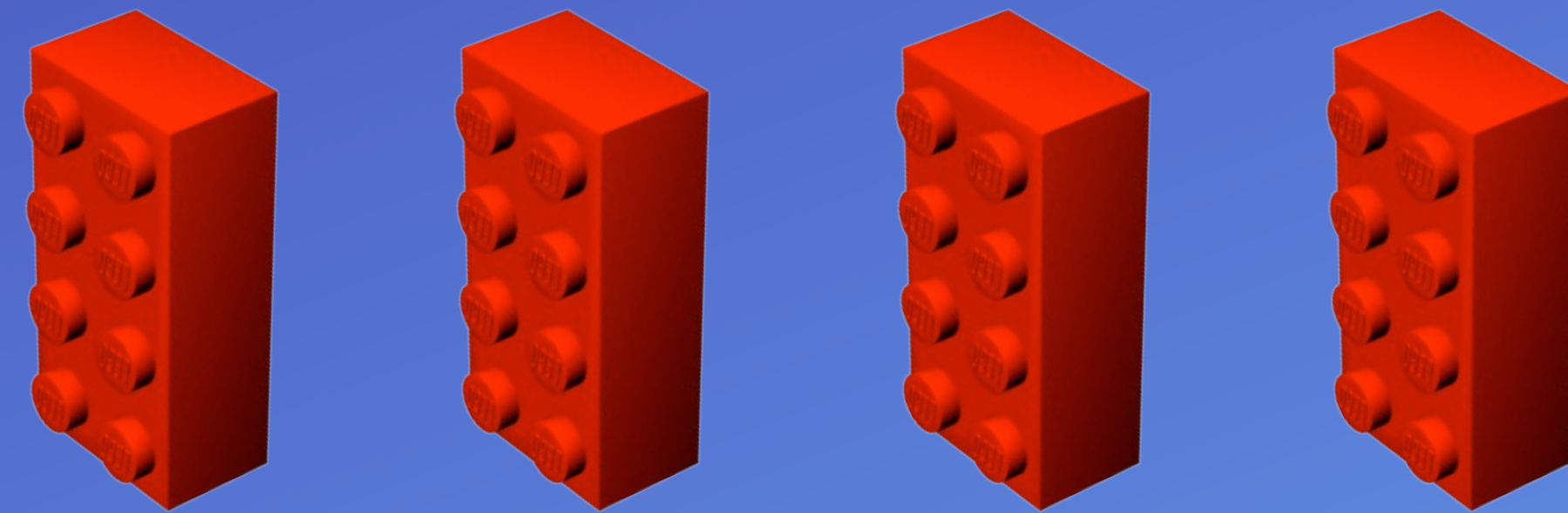
Generic Fold Right



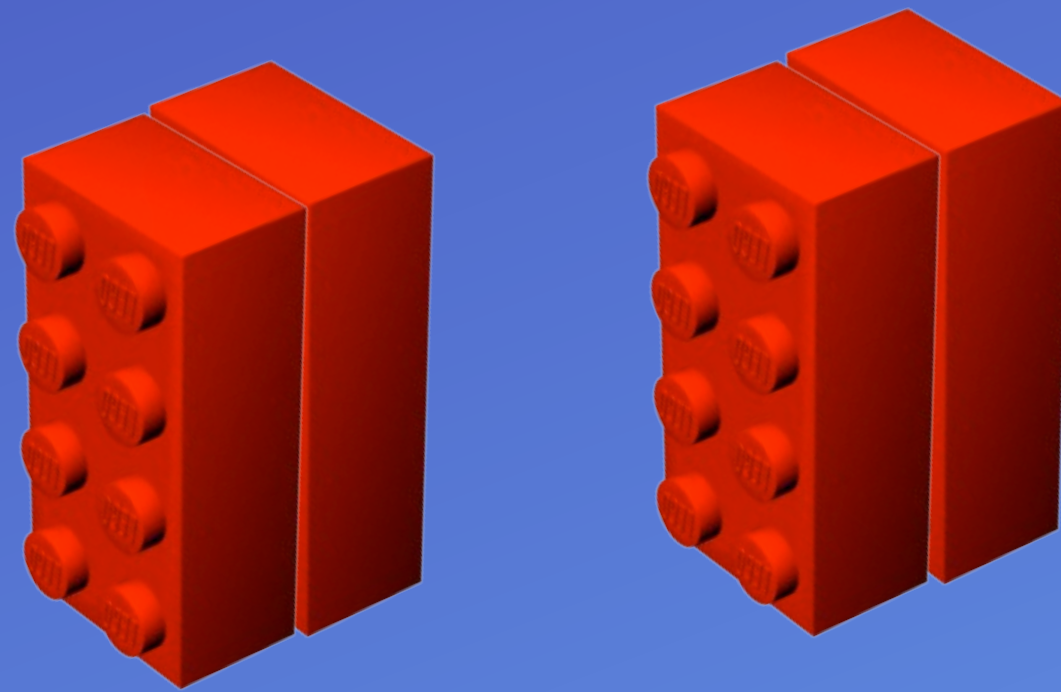
Generic Fold Right



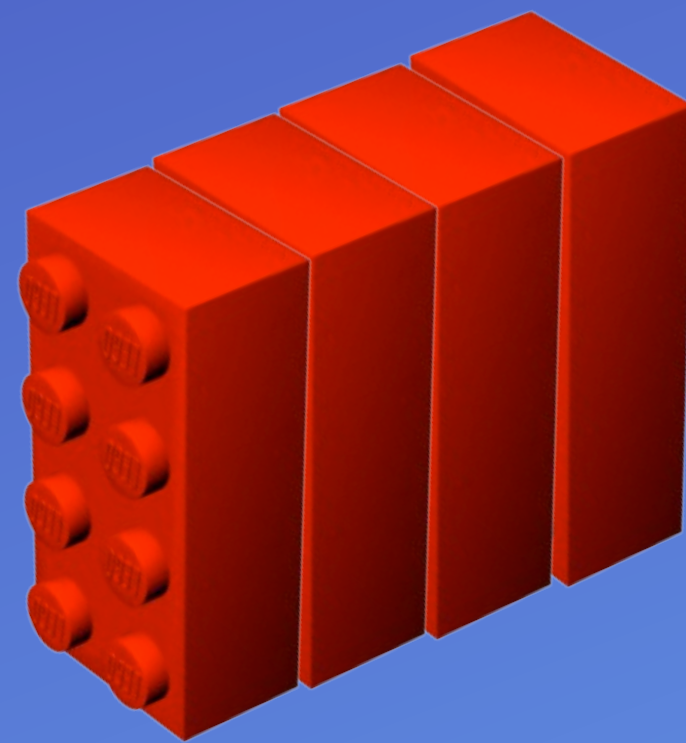
Generic Optimized Fold



Generic Optimized Fold



Generic Optimized Fold



Crazy Monoids

$$(a \rightarrow b) \text{ <> } (b \rightarrow c) == (a \rightarrow c)$$

Can a function type $(a \rightarrow b)$ be a monoid?

What is the operation?

What is the neutral element?

Is it associative?

Function Composition

$\text{co}(f,g) = x \Rightarrow f(g(x));$

$\text{id}(x) = x;$

$\text{co}(\text{id},g) == x \Rightarrow \text{id}(g(x))$ // definition co
 $== x \Rightarrow g(x)$ // apply id
 $== g$ // qed

Functors map

`[1, 2, 3].map(x=> x+2) == [3, 4, 5]`

`(Just 1).map(x=> x+2) == Just 3`

`Nothing.map(whatever) == Nothing`

List, Tree, Stream, Optional, Pipeline, Error,
Validation, Observable, Future, Promise, ..

Functors compose

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

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

```
co( functor.map(f), functor.map(g) )  
  == functor.map( co(f, g) )
```

```
functor.map(id) == id // only for completeness
```

Functors are not Monoids

`[1, 2, 3].map(x => x.toString()) == ["1","2","3"]`

`[Int].map(Int -> String) -> [String]`

`functor a .map(a -> b) -> functor b`



Clever Idea:

instead of $(a \rightarrow b)$

provide a **special mapping** with $(a \rightarrow \text{functor } b)$,
which is essentially a constructor for functors.

$\text{functor } a \quad \langle \rangle \quad \text{functor } b \quad \Rightarrow \quad \text{functor } b$

$\text{functor } a \quad .\text{endo}(a \rightarrow \text{functor } b) \quad \Rightarrow \quad \text{functor } b$

Clever Idea in Action

```
[1, 2, 3].endo( x => replicate(x, x.toString()) ) ==  
  [ ["1"], ["2","2"], ["3","3","3"] ]
```

```
then flatten => ["1", "2", "2", "3", "3", "3"] //aka "join"
```

```
funcA.flatMap(f) = funcB.flatten(funcA.endo(f))
```

```
// aka "bind" or ">>="
```

Finalising the Monoid

`functor a` .flatMap(`a->functor b`) => `functor b`

`(a->functor a) <> (a->functor b) => (a->functor b)`

We need an `(a-> functor a)` `ctor` and flatMap (we already have map, so we only need flatten). If the functor is `monoidal` with flatMap as `<>` and `ctor` as neutral element, then we call it a `Monad`.

Wrapping up

Let $(m\ a)$ be a given monad over type a .

$(m\ a)$ has a ctor $(a \rightarrow m\ a)$. // aka "return" or "pure"

$(m\ a)$ is a functor over a .

$(m\ (m\ a))$ can be flattened to $(m\ a)$.

$(a \rightarrow m\ a).flatMap(a \rightarrow m\ b)$ is a monoidal operation.

Alternative way of writing in pure FP style

$(>>=) :: Monad\ m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Takeaways

Associativity (monoid) requires **pure** functions.

Monoids can **fold** but they cannot escape.

Monads are the most versatile functors (map, filter, expand, reduce) that composes and folds without escaping.

Use Cases

Purely functional state threads

List comprehensions, Streams (possibly reactive)

CompletableFuture, Promises, Continuations

LINQ-style database access

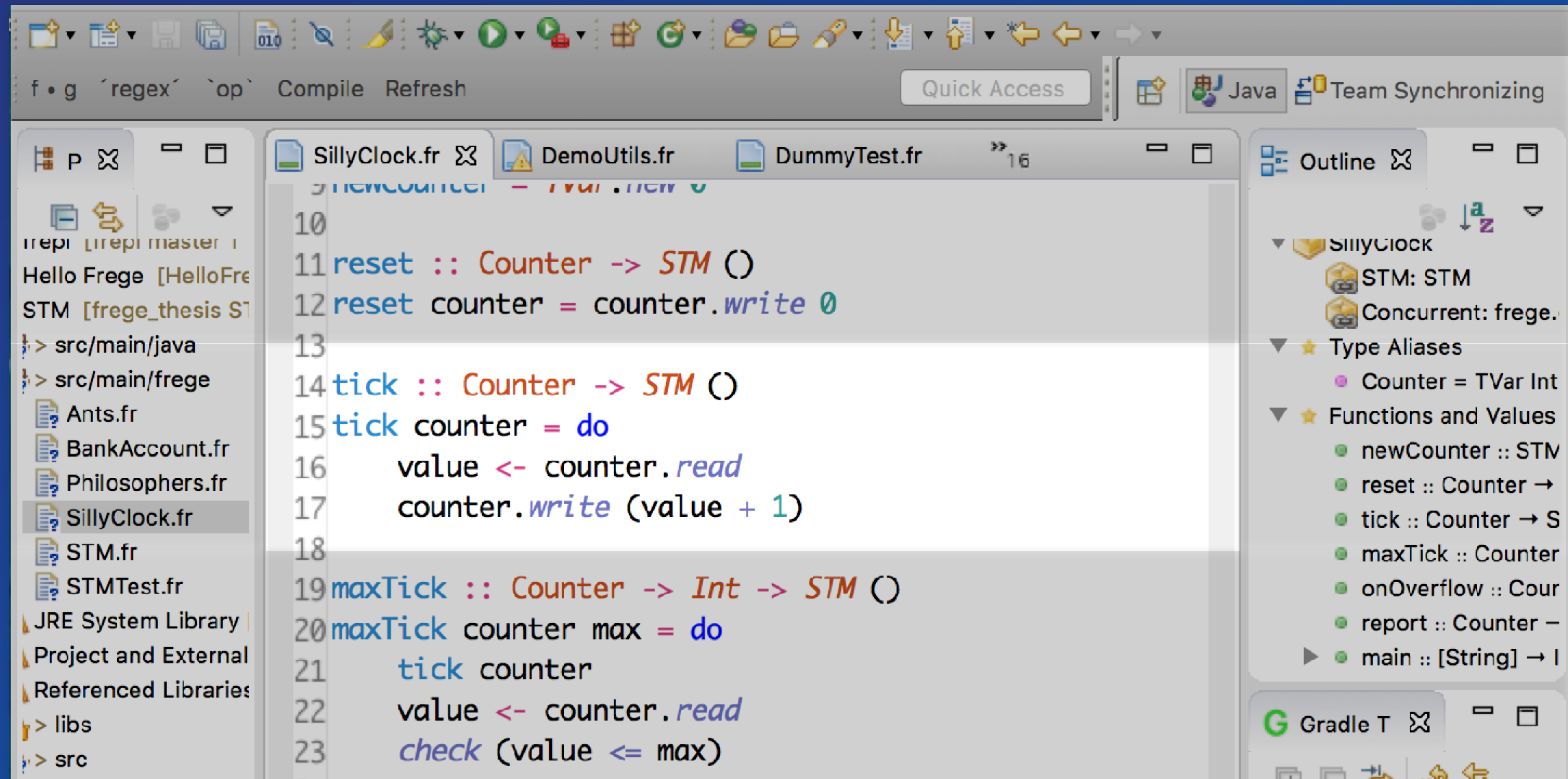
Either, Validation, Exception handling

Optionality, Indeterminism, Parsers, Search trees

Sequencing IO actions, isolating UI actions

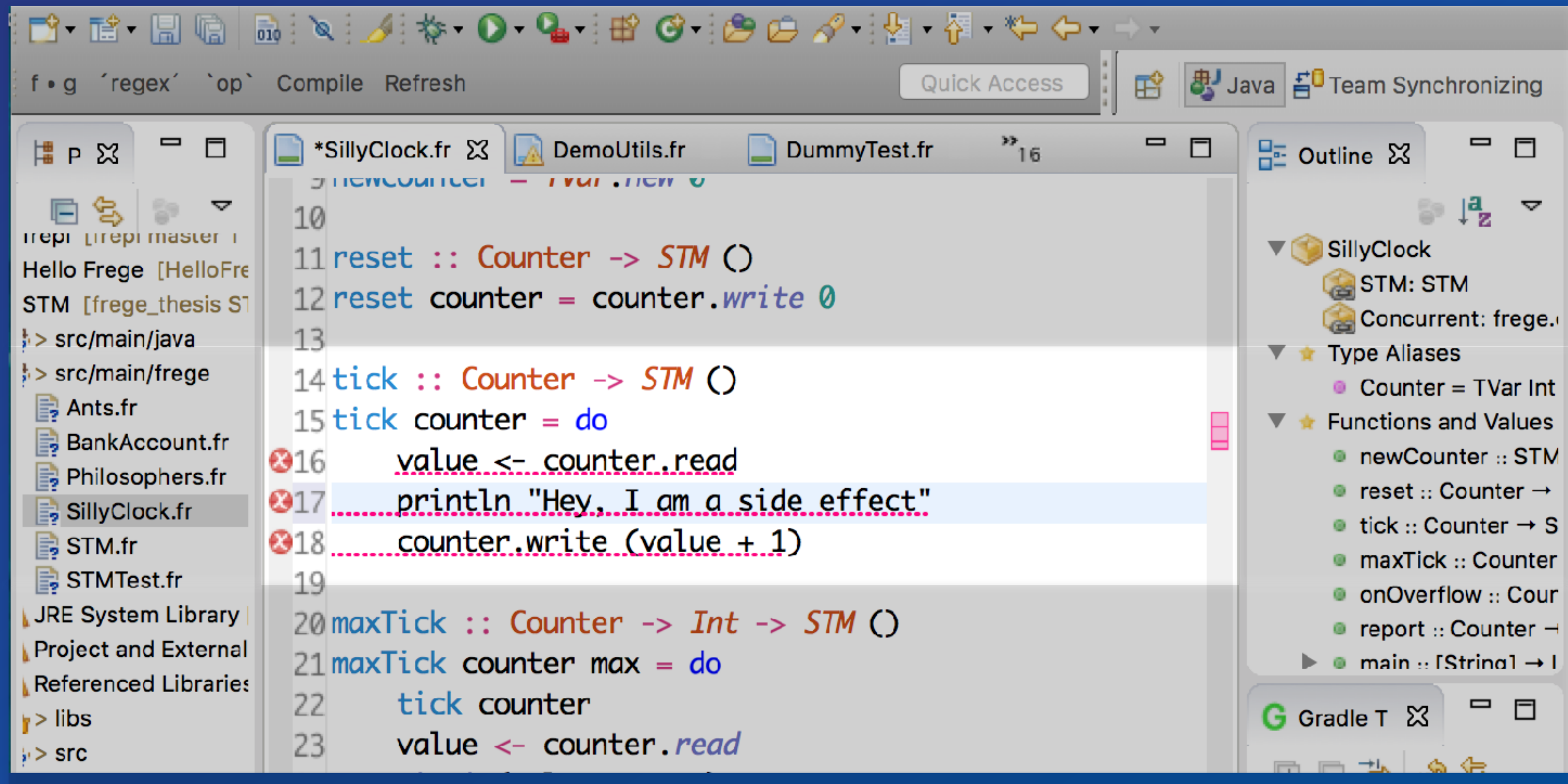
STM transactions, ...

No IO in Transactions!



Type inference FTW

Less tricky
errors



The screenshot shows an IDE with a Scala file named `SillyClock.scala` open. The code defines a `Counter` type alias and several functions: `reset`, `tick`, `maxTick`, and `main`. The `tick` function contains three lines of code that are highlighted with red squiggly lines, indicating type inference errors:

```
11 reset :: Counter -> STM ()
12 reset counter = counter.write 0
13
14 tick :: Counter -> STM ()
15 tick counter = do
16     value <- counter.read
17     println "Hey, I am a side effect"
18     counter.write (value + 1)
19
20 maxTick :: Counter -> Int -> STM ()
21 maxTick counter max = do
22     tick counter
23     value <- counter.read
```

The IDE's left sidebar shows a project structure with files like `Ants.scala`, `BankAccount.scala`, `Philosophers.scala`, `SillyClock.scala`, `STM.scala`, and `STMTest.scala`. The right sidebar shows an outline of the code, including the `Counter` type alias and the `tick` function. The bottom status bar shows the Gradle tool.

There is a world...

... where logical reasoning rules and structure arises from consistency and a rich set of relations.

Exploring this world is like programming without implementation.

Purely functional programming opens the door.
Consider **Haskell**, **Frege**, **Purescript**, **Idris**.

Please give feedback!

Prof. Dierk König
canoo

