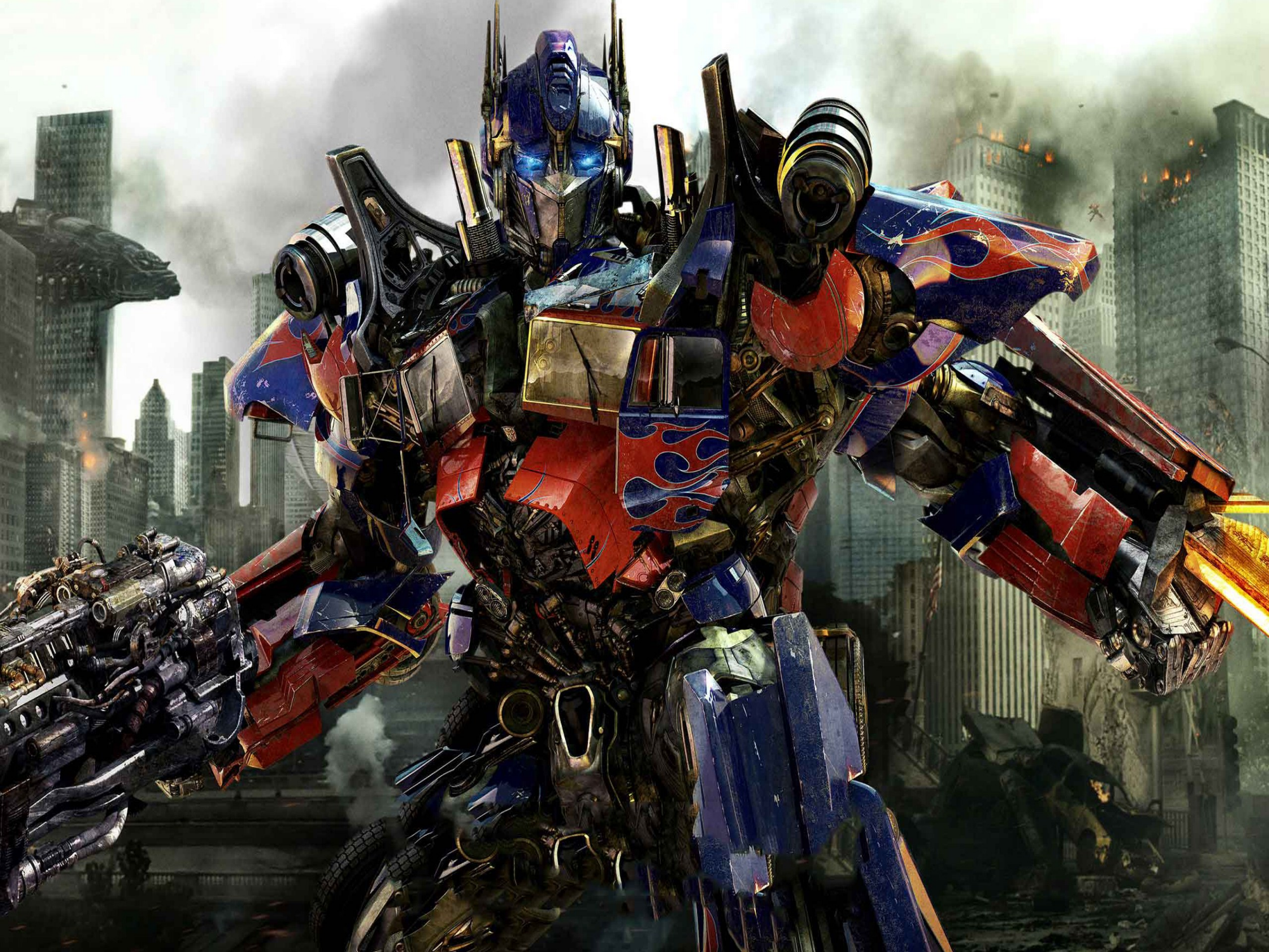
OptimusPrimeT

Lars Hupel

April 27th, 2015



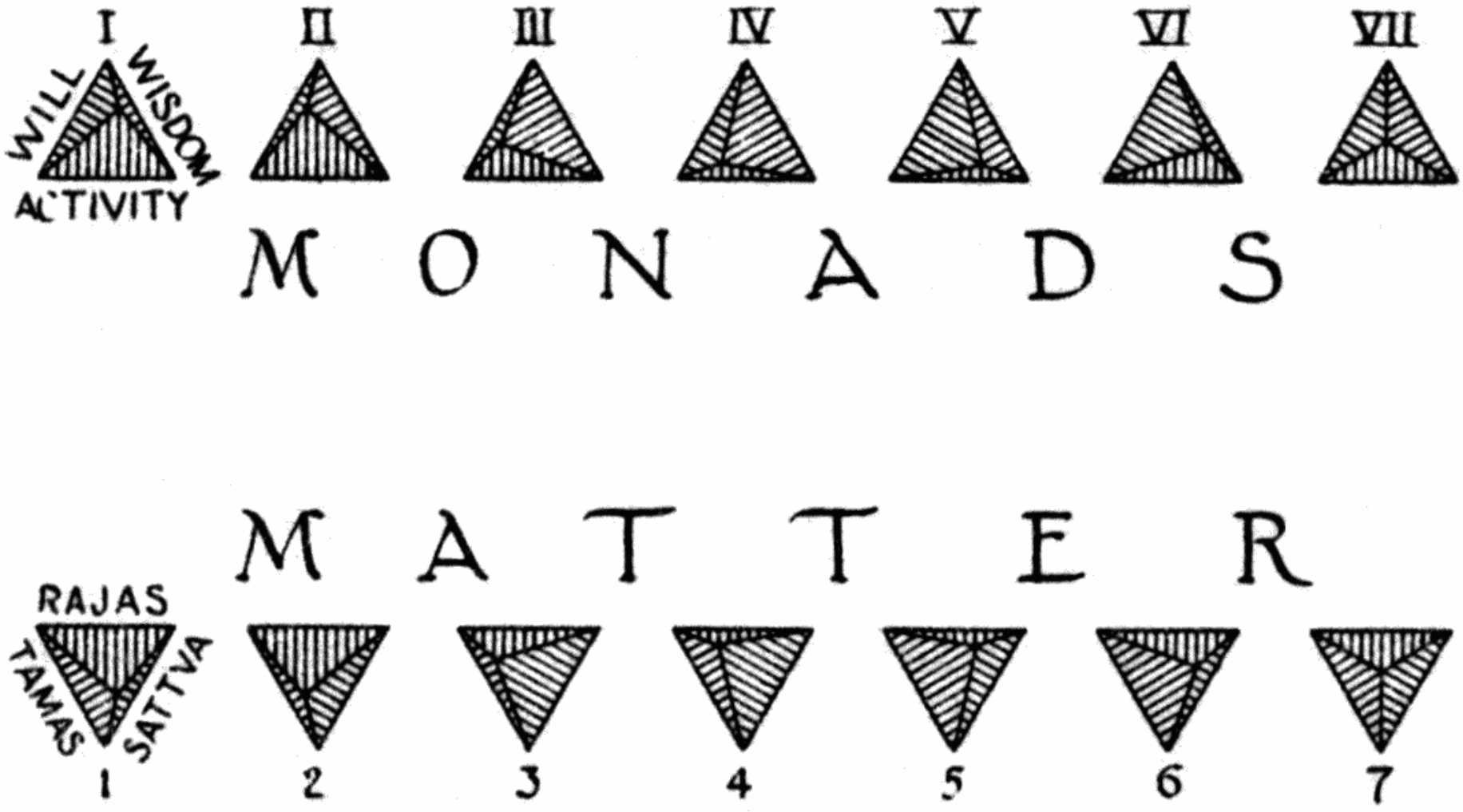
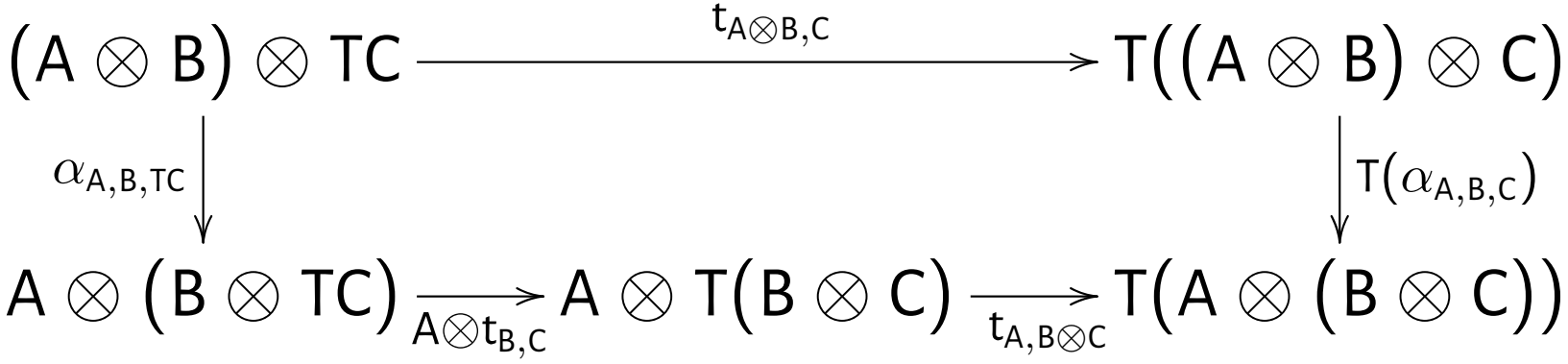
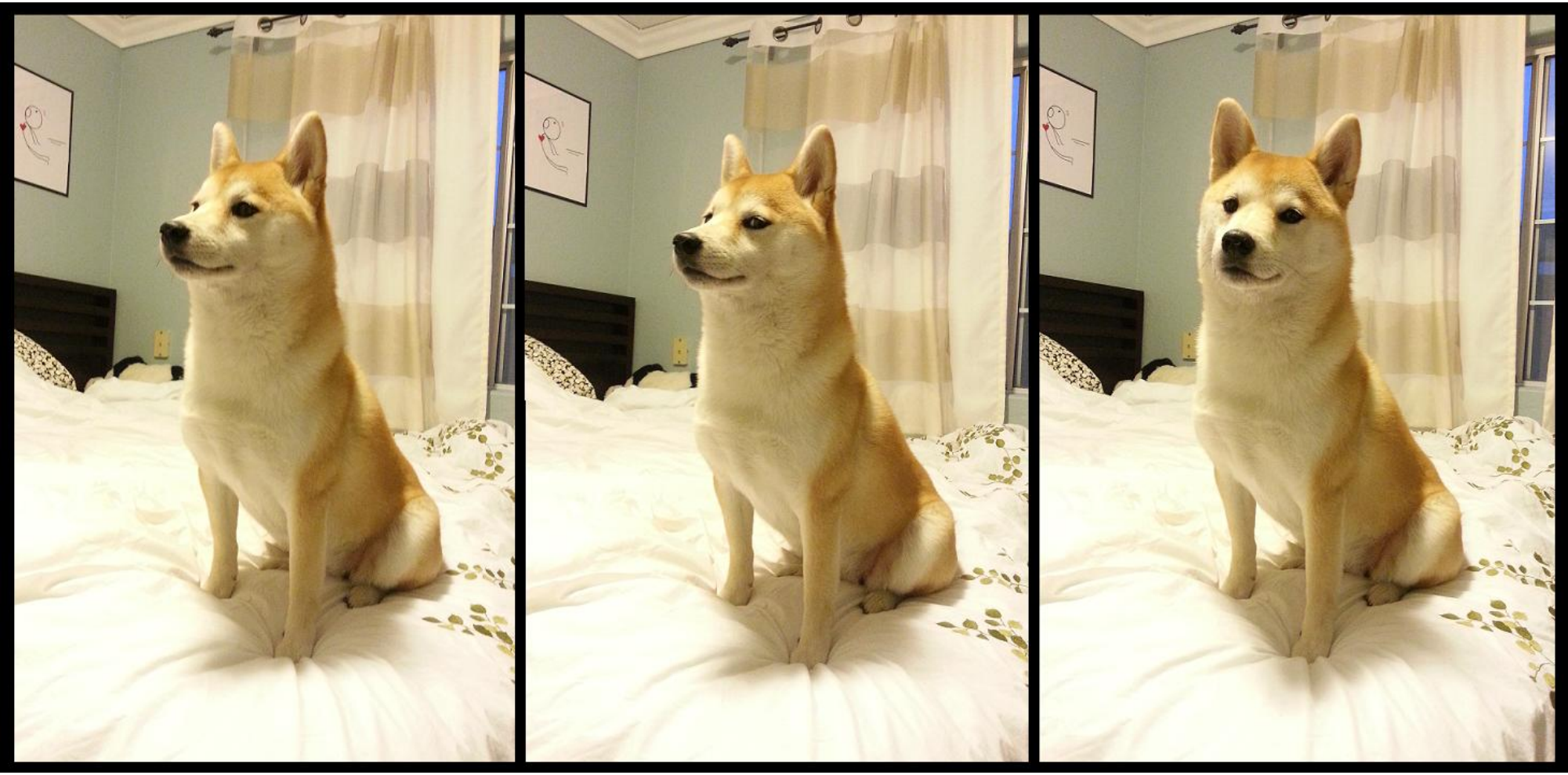


DIAGRAM VIII.—The Seven Types of Monads and the Seven Types of Matter.


$(A \otimes B) \otimes TC$
$t_{A\otimes B,C}$
$T((A \otimes B) \otimes C)$
$\alpha_{A,B,TC}$
$T(\alpha_{A,B,C})$
$A \otimes (B \otimes TC)$
$A\otimes t_{B,C}$
$A \otimes T(B \otimes C)$
$t_{A,B\otimes C}$
$T(A \otimes (B \otimes C))$




TAKE EFFECTS
FROM ONE MONAD
PUT THEM INTO
ANOTHER MONAD
imgflip.com

Agenda



1. Functors, Applicatives, Monads
2. Composition of Functors
3. Monad Transformers

Functors, Applicatives, Monads



Everyone knows **monads**, right?

But: there's more.

If all you have are monads ...



Monads have lots of use cases:

- error handling
- parser combinators
- value-oriented parallelism
- ...

Limitations of monads



Code using monads has both mathematical and practical limitations



Demo$_1$

Functors from the ground up



Functors from the ground up



Functors are about **transforming** data

```
scala> List(1, 2, 3).map(x => x + 1)
List(2, 3, 4)

scala> Some(1).map(x => x + 1)
Some(2)

scala> Future { 1 }.map(x => x + 1)
// something resembling 2
```

The `Functor` class



`cats` offers a generalization of that pattern

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

More power: Applicatives



Applicatives are about **transforming independent** pieces of data

```
scala> Applicative[List].
         map2(List(1, 2), List(3, 4))(_ -> _)
List((1, 3), (1, 4), (2, 3), (2, 4))
```

The `Applicative` class



```
trait Applicative[F[_]] extends Functor[F] {
  def map2[A, B, C](fa: F[A], fb: F[B])
                   (f: (A, B) => C): F[C]
}
```

Even more power: Monads



Monads are about **transforming dependent** pieces of data

```
def fileRequest(user: User, file: Path): Future[Int] =
  for {
    token <- db.getAccessToken(user)
    input <- db.readFile(token, file)
    _ <- HTTP.sendHeader(HTTP.OK)
    written <- HTTP.sendBody(input)
  } yield written
```

The Monad class



```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

The Monad class



```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

  def map2[A, B, C](fa: F[A], fb: F[B])
                   (f: (A, B) => C): F[C] =
    ???
}
```

The Monad class



```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

  def map2[A, B, C](fa: F[A], fb: F[B])
                   (f: (A, B) => C): F[C] =
    flatMap(fa) { a =>
      flatMap(fb) { b =>
        f(a, b)
      }
    }
}
```

`Monad` vs `Applicative` revisited



Demo$_2$

Agenda



1 Functors, Applicatives, Monads

2 Composition of Functors

3 Monad Transformers

Programming is ...



Abstracting!

But: **non-composable** abstractions are useless

Composing functors



Input

```
def intToString(x: Int): String
val xss: List[List[Int]]
```

Output

```
val yss: List[List[String]] = ???
```



Demo$_3$

But what about monads?



Surely we can also compose monads.

But what about monads?



Surely we can also compose monads.

Except we can't

But what about monads?



Surely we can also compose monads.

Except we can't (in general)

Agenda



1. Functors, Applicatives, Monads
2. Composition of Functors
3. Monad Transformers

Composing monads



Our Tools

```
// Monad[F]
def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

// Monad[G]
def flatMap[A, B](fa: G[A])(f: A => G[B]): G[B]
```

Our Goal

```
def flatMap[A, B](fga: F[G[A]])
                  (f: A => F[G[B]]): F[G[B]]
```

Computer Science Interlude



The Halting Problem

It is undecidable whether a program will terminate on a given input.

Computer Science Interlude



The Halting Problem

It is undecidable whether a program will terminate on a given input.

… defeated?

Yet there are programming languages which only accept terminating programs.

Composing particular monads



Our Tools (refined)

```
// Monad[F]
def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

Our Goal (refined)

```
def flatMapForOption[A, B]
  (foa: F[Option[A]])
  (f: A => F[Option[B]]): F[Option[B]]
```



Demo$_4$

When do monads transform?



It's difficult.

Recap



- Monads are awesome, but often too powerful.
- Functors compose.
- Applicatives compose.
- Monads do not compose in general.
- Specific monads do compose.
- That actually comes up very often.







The End

@larsr_h
larsrh.github.io
typelevel.org/blog