Deconstructing Monads

The joy of the double exponential

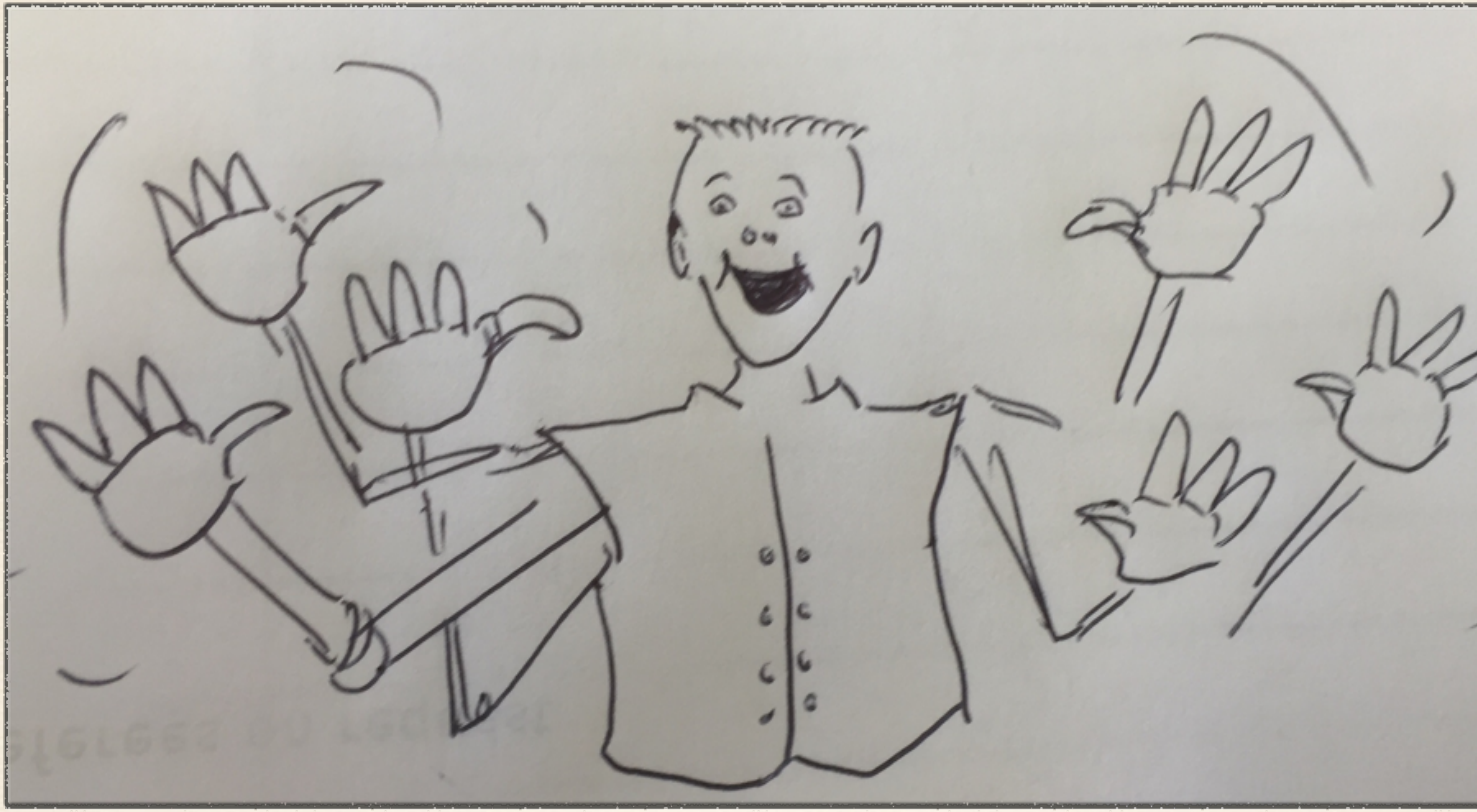
Monads and double exponentials

- Recap on monads
- Every monad* is a double exponential $\boldsymbol{\beta}_H$ for some H
- For any H, $\boldsymbol{\beta}_H$ deconstructs H for you
- For suitably chosen H, $\boldsymbol{\beta}_H$ = an inexhaustible treasure trove of math
- My plan: Use Bewl to calculate $\boldsymbol{\beta}_\neg$ to solve parity
- $\neg$ ("not") interchanges true and false. It's an object (dot) in the topos of permutations

* terms and conditions apply

Math is not for everyone

So…



A maddeningly vague, hand-wavy explanation

seemed better than…

…going into too much detail

although I still want to be like him

Monads explained in one slide

- In Scala, classes like **List**, **Option** and **Future** can be used in *for comprehensions*
- This is because they have certain features in common
- For example, you can collapse a **List[List[X]]** into a **List[X]**
- Same with the others. They are all *monads*

What is actually happening when you collapse a List[List[X]] ?

- It's a special case of collapsing a quadruple exponential into a double one:

```
def collapse[X, H]: (
    (((X => H) => H) => H) => H
) => (
     (X => H) => H
) =
  xhhhh => xh => xhhhh(_(xh))
```

- because there is a type H such that **List[X]** = (**X => H**) **=> H**

- i.e. **List** = $\boldsymbol{\beta}_H$

Singleton lists and list map are captured, too

- In the same spirit, we can define **singleton** and **map** operations for double exponentials, which correspond to the same operations on lists:

```
def singleton[X, H]: X => (
  (X => H) => H
) =
  x => xh => xh(x)

def map[X, Y, H]: (
  X => Y
) => (
  ((X => H) => H) => ((Y => H) => H)
) =
  xy => xhh => yh => xhh(x => yh(xy(x)))
```

- These operations **singleton, map, collapse** obey the monad laws
- This all explains why **List, Option** etc have the properties they do

Monads describe structure

- There are natural* functions
```
List[List[X]] => List[X]
List[Int] => Int
List[X => X] => (X => X)
```
- This is because **List[X]**, **Int** and **X => X** are all algebras over **List**
- There's a concept of M-algebras for any monad M
- Actually, **List-**algebras are just monoids

* i.e. obeying modified versions of the monad laws

Fundamental theorem

For any* object H in a* category,

- $\boldsymbol{\beta}_H$: X => ((X => H) => H) is a monad
- H is a $\boldsymbol{\beta}_H$-algebra
- For any other monad M, M-algebra structures on H are interchangeable with arrows M => $\boldsymbol{\beta}_H$.

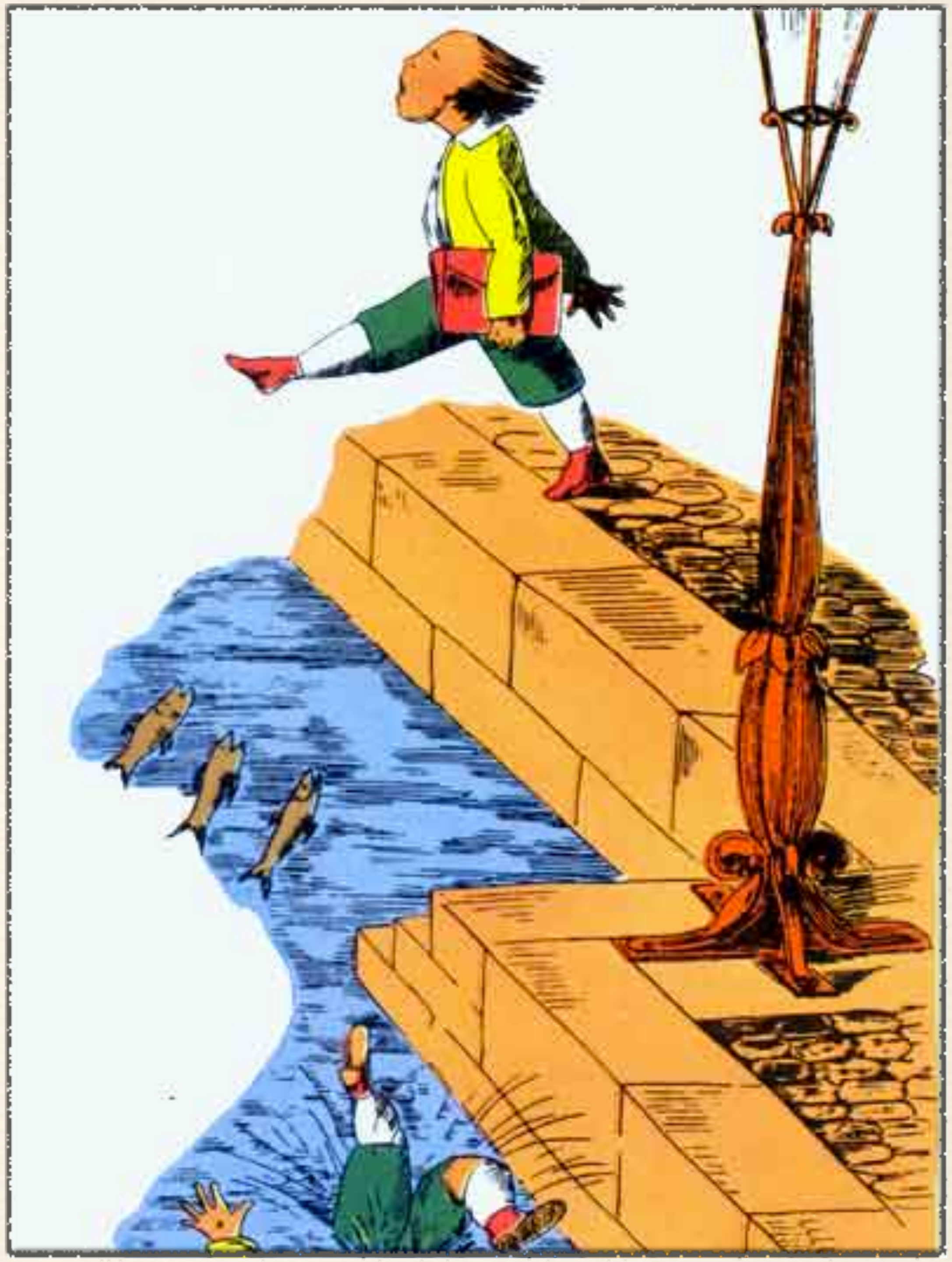
So $\boldsymbol{\beta}_H$ precisely captures all the algebraic structure H could ever have over any conceivable monad.

* suitably structured

Ascending to a higher level of abstraction

Whole branches of math can be described as the study of M-algebras for some monad M, i.e. the study of $\boldsymbol{\beta}_H$-algebras for some object H.

$\boldsymbol{\beta}_{<2,\wedge,\vee,\neg>}$ is topology
$\boldsymbol{\beta}_{[0,\ 1]}$ is probability measure theory
$\boldsymbol{\beta}_{\text{unit circle}}$ is Fourier analysis



You can't just study β_H for any old H

- β_0 and β_1 are trivial
- β_2 is already so complicated that there is a page on Wikipedia describing its structure: https://en.wikipedia.org/wiki/Post%27s_lattice
- There are results about β_3
- For β_4 etc they've given up

- I think it should be possible to structure the simplest possible permutation **¬** = (true, false) so that **β**¬ is manageable (and possibly leads to an amazing new theory).

- Initial results suggest that | **β**¬(n)| = 2n.
  A promising start

- Meanwhile here is the Bewl code for double-exponential monads:

It’s test-driven, of course

```
val monadJoin = omega.doubleExpMonad

describe("The double-exponential monad can be constructed for
sets, and...") {
  it("values at a dot are cached") {
    (monadJoin(O) eq monadJoin(O)) shouldBe true
  }

  it("free objects have the right size") {
    monadJoin(O).free.globals should have size 2
    monadJoin(I).free.globals should have size 4
    monadJoin(omega).free.globals should have size 16
  }

  it("embedding (eta) works") {
    val eta: Symbol > (Symbol → TRUTH → TRUTH) =
monadJoin(two).eta
    for (
      f <- elementsOf(two > omega);
      symbol <- Seq('x, 'y)
    )
      eta(symbol)(f) shouldBe f(symbol)
  }

  it("functoriality (map) works") {
    val symbols = dot('a, 'b)
    val ints = dot(1, 2, 3)
    val f: Symbol > Int = arrow(symbols, ints, 'a -> 2, 'b ->
1)
    val map: (Symbol → TRUTH → TRUTH) > (Int → TRUTH → TRUTH) =
monadJoin.map(f)

    // TODO: abstract away 'io' here
    for (
      soo <- elementsOf(symbols > omega > omega);
      io <- elementsOf(ints > omega);
      symbol <- Seq('a, 'b)
    )
      map(soo)(io) shouldBe soo((omega > f) (io))
  }
```

In **BaseTopos**, a **Monad** trait

```
trait Monad[M[X <: ~] <: ~] {
  final private val memoizedLocalValues =
    Memoize.generic.withLowerBound[
      DOT,
      At,
      ~
    ] (atUncached)

  final def apply[X <: ~](dot: DOT[X]): At[X] =
    memoizedLocalValues(dot)

  def atUncached[X <: ~](dot: DOT[X]): At[X]
  def map[X <: ~, Y <: ~](arrow: X > Y): M[X] > M[Y]

  trait At[X <: ~] {
    val free: DOT[M[X]]
    val eta: X > M[X]
    val mu: M[M[X]] > M[X]

    def sanityTest = {
      mu o map(eta) shouldBe free.identity
      mu o apply(free).eta shouldBe free.identity
    }

    def sanityTest2 = {
      mu o map(mu) shouldBe (mu o apply(free).mu)
    }
}
```

Implementing β_H

```
lazy val doubleExpMonad =
  new Monad[
    ({type λ[X <: ~] = X → S → S}) # λ
  ] {
    override def atUncached[X <: ~](
      dash: DOT[X]
    ) =
      new At[X] {

        private lazy val doubleExp: EXPONENTIAL[X → S, S] =
          dash > dot > dot

        override lazy val free: DOT[X → S → S] =
          doubleExp

        override lazy val eta =
          doubleExp.transpose(dash) {
            (x, f) => f(x)
          }

        override lazy val mu =
            (dash > dot > dot).transpose(
              dash > dot > dot > dot > dot
            ) {
              (ffff, f) => ffff(
                (dash > dot > dot > dot).transpose(
                  dash > dot
                ) {
                  (x, f) => f(x)
                }(
                  f
                )
              )
            }
      }

    override def map[X <: ~, Y <: ~](
      arrow: X > Y
    ) =
      dot > (dot > arrow)
  }
```

"Mathematics is a game played according to certain simple rules with meaningless marks on paper."

–David Hilbert

THANK YOU

http://github.com/fdilke/bewl