http://github.com/fdilke/bewl

Universal Algebra in Bewl

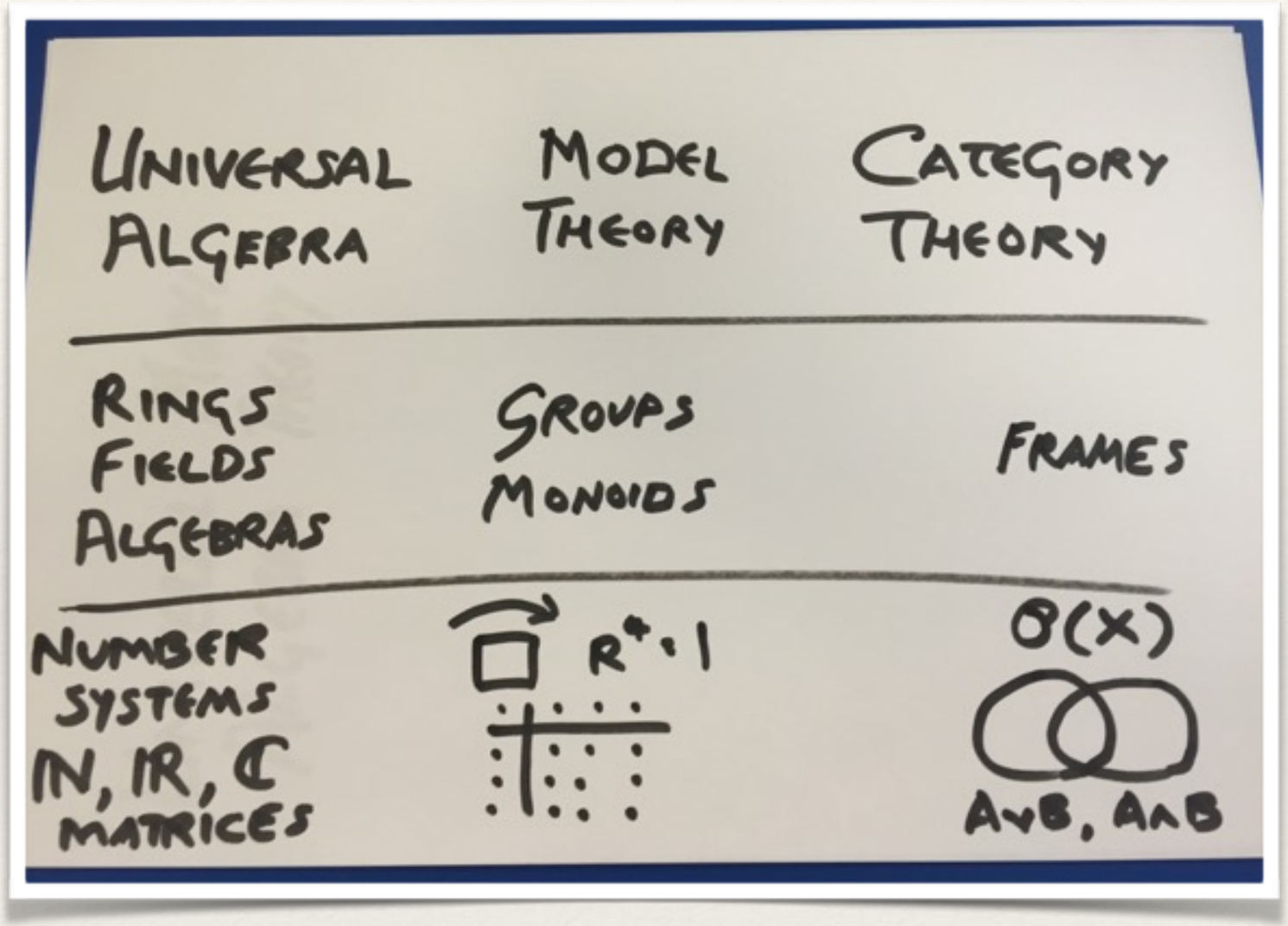
Enhancing the DSL so it can slice and dice algebraic structures

Algebraic structures? Slice and dice? Why?

- What is an "algebraic structure"?
  Examples: monoid, group, ring, vector space, etc etc
- What slicing and dicing can you do with them?
  Define algebraic theories
  Verify properties of algebras and algebra maps
  Combining, extending and manipulating them to create new ones
- Why?
  To speed up Bewl so I can apply it to music theory

The view from 3,000,000 feet

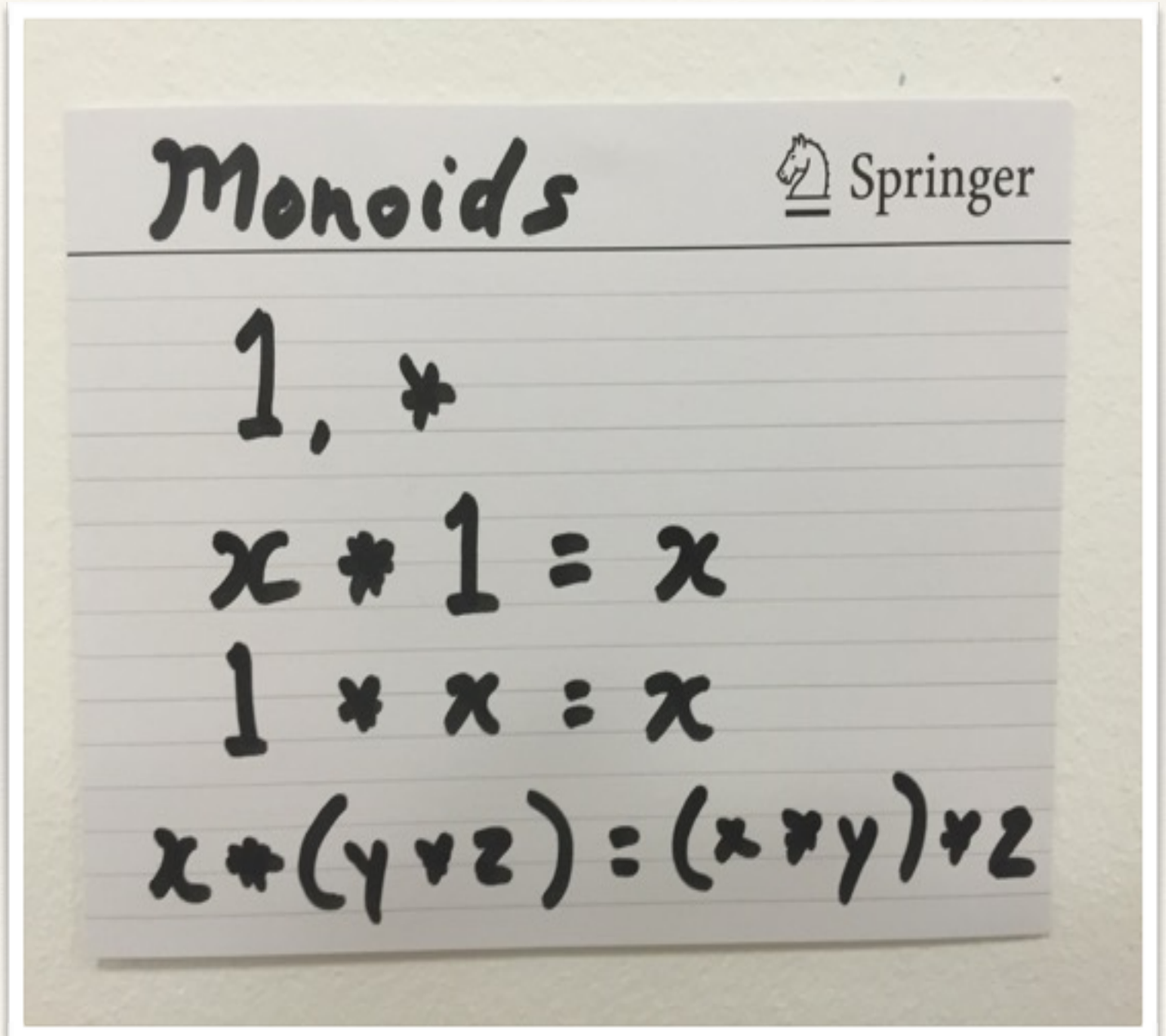
- Theories of theories
- Algebraic theories
- Workspaces where people do actual real calculations



What is an algebraic theory?

- A whole branch of maths defined on the back of a postcard

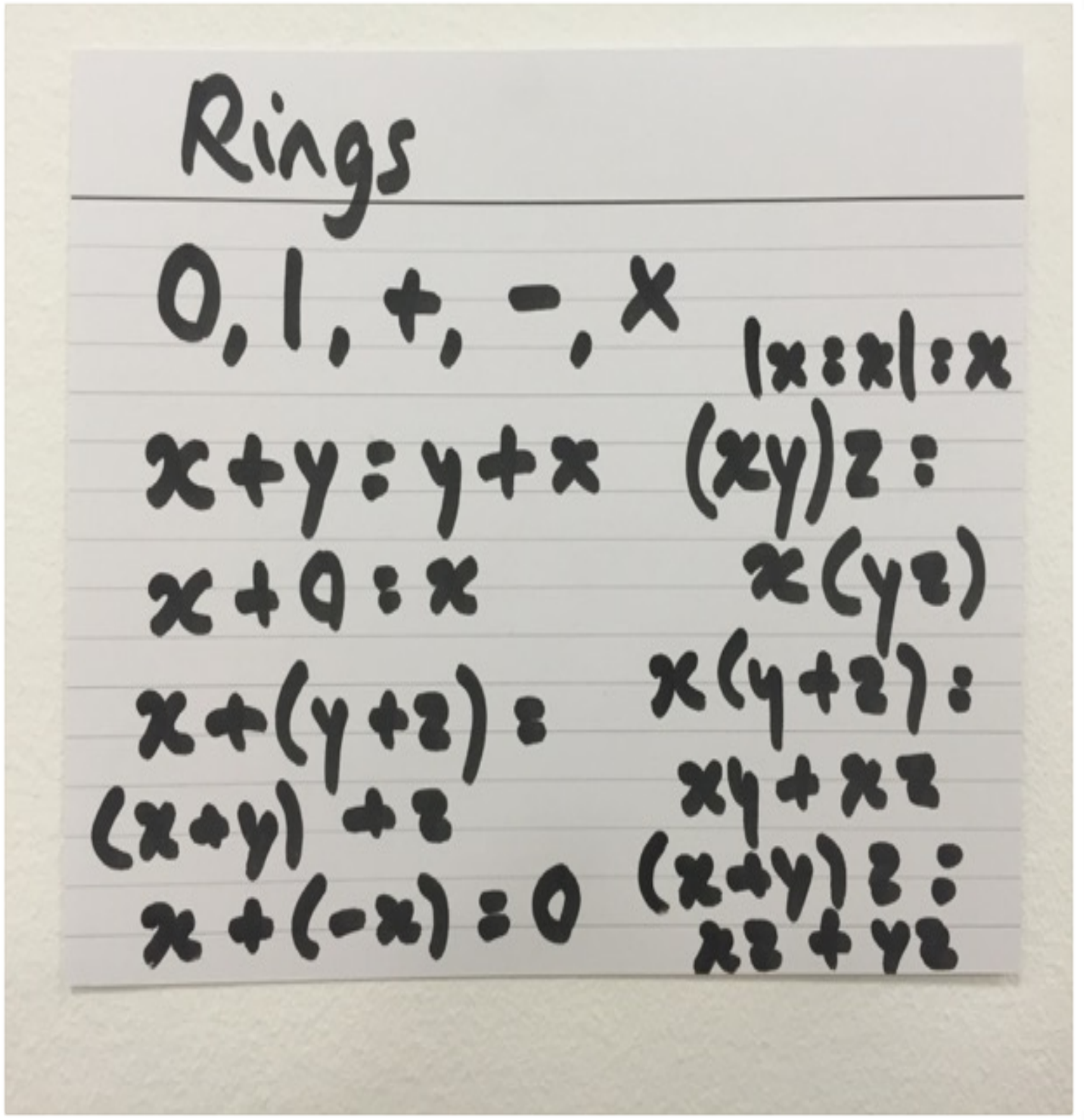
- Example: Monoids



Another example

Rings

Examples of rings:
Integers (**Z**)
Rational numbers (**Q**)
Real numbers (**R**)
Complex numbers (**C**)
Quaternions (**H**)
Matrices
etc etc etc



How did I do this in Java?

- Bytecode modification
- javassist
- Annotations
- Group can extend Monoid

```
// Axioms for a set with a associative binary operation
@Axioms
public abstract class Monoid<T extends Element<T>> implements
Algebra<Monoid<T>> {
  @Operator public abstract T unit();
  @Operator public abstract T multiply(T x, T y);

  @Law public boolean leftUnit(T x) {
    return equality(
      multiply(unit(), x),
      x
    );
  }

  @Law public boolean rightUnit(T x) {
    return equality(
      multiply(x, unit()),
      x
    );
  }

  @Law public boolean associative(T x, T y, T z) {
    return equality(
      multiply(x, multiply(y, z)),
      multiply(multiply(x, y), z)
    );
  }
```

How did I do this in Clojure?

```
(define-law commutative-law [op] [x y]
  (= (op x y) (op y x))
)

(def monoid {:signature { :unit 0 :multiply 2 }   ; a binary product with 2-sided unit
             :laws [    (unit-law :unit :multiply)
                        (associative-law :multiply)
]})

(def group (extend-theory monoid            ; a monoid with inverses
           {:signature { :inverse 1 }
            :laws [ (inverse-law :unit :inverse :multiply)
]}))
```

- Macro “define-law”
- Compact, expressive
- No type checking to worry about

How do the constructions look?

```
(defn group-of-units
  "The group of units of a monoid, and its embedding morphism"
  [MM] (let [
      {:keys [carrier unit multiply]} MM
      topos (topos-of carrier)
      {:keys [truth]} topos
      [the-1 to-1] (topos :terminator)
      M (MM :carrier)
      eq-M (equals-over M)
      [MxM _ [pi1 pi2] mulMxM :as square] ((topos :product) [M M])
      [G embed factorize] (subobject [m-n MxM] (let [
        m (pi1 m-n)
        n (pi2 m-n)
        unit-src (unit (to-1 (m :src)))
        ] ((truth :and) (eq-M unit-src (multiply m n))
                        (eq-M unit-src (multiply n m)))
      ))
      extract (fn [m-n] (let [e (embed m-n)] [(pi1 e) (pi2 e)]))
    ] [{ :theory group :carrier G
       :unit (factorize (mulMxM [unit unit]))
       :multiply (build-multiary [m-n G p-q G] (let [
          [m n] (extract m-n)
          [p q] (extract p-q)
       ] (factorize (mulMxM [(multiply m p) (multiply q n)])))
      ))
        :inverse (build-multiary [m-n G] (let [
         [m n] (extract m-n)
        ] (factorize (mulMxM [n m]))
      ))} (pi1 embed) ]
))
```

- Even simple constructions are quite awkward

In Scala, it should all be easier because…

- Strong typing; support for higher-kinds, traits, abstract types
- If your program compiles, it probably works
- Underlying Bewl DSL is much more expressive

So let's be more ambitious:

- Music theory requires algebraic theories that refer to already existing theories in their definition
- Extend the grammar to handle 'extension by scalars'

Still, there are challenges

- The IDE has long ceased to be able to make sense of Bewl
- It's too difficult to keep the type checker happy
- But, I am getting there. You can now define an algebraic theory in one line of code!

```
 val commutativeMagmas = AlgebraicTheory()(∗)(α ∗ β := β ∗ α)
 case class CommutativeMagma[T <: ~](
   carrier: DOT[T], op: BinaryOp[T]
 ) extends commutativeMagmas.Algebra[T](carrier)(∗ := op)

 val carrier = dot(true, false)
 val commutativeOp = bifunctionAsBiArrow(carrier) { _ & _}
 val nonCommutativeOp = bifunctionAsBiArrow(carrier) { _ & !_ }

intercept[IllegalArgumentException] {
   new commutativeMagmas.Algebra[Boolean](carrier)().sanityTest
}
intercept[IllegalArgumentException] {
   new commutativeMagmas.Algebra[Boolean](carrier)($plus := commutativeOp).sanityTest
}
```

For more, come to Tom's Category Theory group (every Tuesday)

Thank you