The Axes of Abstraction

"Give me six hours to develop an abstraction and I will spend the first four sharpening the axis."

Paul Phillips
Lambdaconf 2017
Boulder, CO

Situation

— Trend: axioms, logic, inference, laws
— Math abstractions (e.g. monoids) succeeding
— Domain abstractions remain mostly ad hoc
— "More art than science"
— That's a euphemism for "not science"

Thesis

There is science we can apply.

We can constrain more.
We can infer more.
We can derive more.
We can resist **adhocicity**.

What is with "axes"

— In the Cartesian sense: x-axis, y-axis, z-axis
— Complex numbers literally add an axis
— Loosely: a property which varies between extrema
— Orthogonality preferred but not required

That didn't sound right...

— ...because abstraction is something else
— That was an axis of **generalization**
— Abstraction vs. generalization, coming up
— First, the cutting room floor

The axis of axis exoticity

I initially thought I'd talk about axes like:

```
- Parametricity       # giver of free theorems
- Arity               # size polymorphism
- Orderity            # kind polymorphism
- Fixity              # operator associativity
- Opacity             # "boxity" says @xeno-by
- Inductivity         # data vs. codata
```

Algebraic axes

Or algebraic properties like:

```
- Associativity          # category theory
- Commutativity          # CRDTs
- Distributivity         # belief propagation
- Idempotency            # lattices
- Invertibility          # isomorphisms
- Selectivity            # idempotent-er
```

Yet more exoticity

Other candidates:

```
- Longevity             # Lifetimes
- Evitability           # evaluation strategy
- Classity              # "first class" is like "power"
- Quotidity             # macro depth polymorphism
```

Forget it!

Instead, I learned about something new.

I think it deserves a wider audience.

Coming up.

But first...

Generalization by Example

Generalization

```
def f(x: 5): 10                    = x + x  // it's 10!
```

Generalization

```
def f(x: 5): 10                    = x + x  // it's 10!
def f(x: Int): Int                 = x + x  // Don't assume 5
```

Generalization

```
def f(x: 5): 10                    = x + x  // it's 10!
def f(x: Int): Int                 = x + x  // Don't assume 5
def f[A <: Number](x: A): A        = x + x  // Don't assume integer
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A  = x + x  // Don't assume numbers
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
def f[A: Semigroup](x: A): A             = x + x  // Don't assume + has identity
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
def f[A: Semigroup](x: A): A             = x + x  // Don't assume + has identity
def f[A: Magma](x: A): A                 = x + x  // Don't assume + is associative
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
def f[A: Semigroup](x: A): A             = x + x  // Don't assume + has identity
def f[A: Magma](x: A): A                 = x + x  // Don't assume + is associative
def f[A](x: A): A                        = x      // Assume nothing
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
def f[A: Semigroup](x: A): A             = x + x  // Don't assume + has identity
def f[A: Magma](x: A): A                 = x + x  // Don't assume + is associative
def f[A](x: A): A                        = x      // Assume nothing
```

Generalization

```
def f(x: 5): 10                          = x + x  // it's 10!
def f(x: Int): Int                       = x + x  // Don't assume 5
def f[A <: Number](x: A): A              = x + x  // Don't assume integer
def f[A: CommutativeRing](x: A): A       = x + x  // Don't assume numbers
def f[A: Ring](x: A): A                  = x + x  // Don't assume * is commutative
def f[A: Rng](x: A): A                   = x + x  // Don't assume * has identity
def f[A: CommutativeGroup](x: A): A      = x + x  // Don't assume * exists
def f[A: Group](x: A): A                 = x + x  // Don't assume + is commutative
def f[A: Monoid](x: A): A                = x + x  // Don't assume + has an inverse
def f[A: Semigroup](x: A): A             = x + x  // Don't assume + has identity
def f[A: Magma](x: A): A                 = x + x  // Don't assume + is associative
def f[A](x: A): A                        = x      // Assume nothing
```

Abstraction v. Generalization

Abstraction

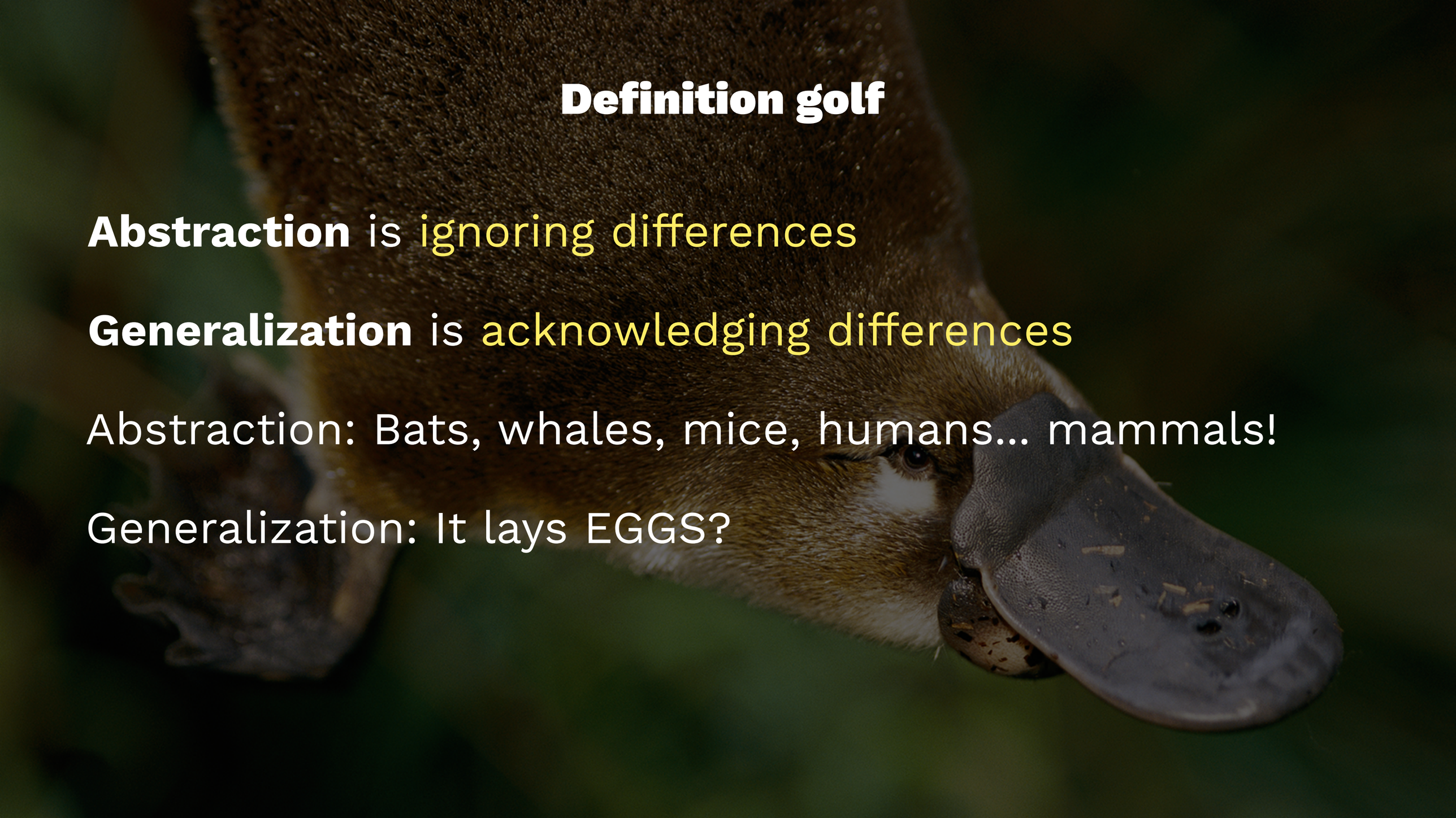
SAGREDO: This is a marvelous idea! It suggests that when we try to understand nature, we should look at the phenomena as if they were messages to be understood. Except that each message appears to be random until we establish a code to read it. This code takes the form of an abstraction, that is, we choose to ignore certain things as irrelevant and we thus partially select the content of the message by a free choice.

-- Gödel, Escher, Bach: an Eternal Golden Braid

Someone else's definitions, my emphasis:

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail).

Generalization is a broadening of application to encompass a larger domain of objects of the same or different type.



Definition golf

Abstraction is ignoring differences

Generalization is acknowledging differences

Abstraction: Bats, whales, mice, humans… mammals!

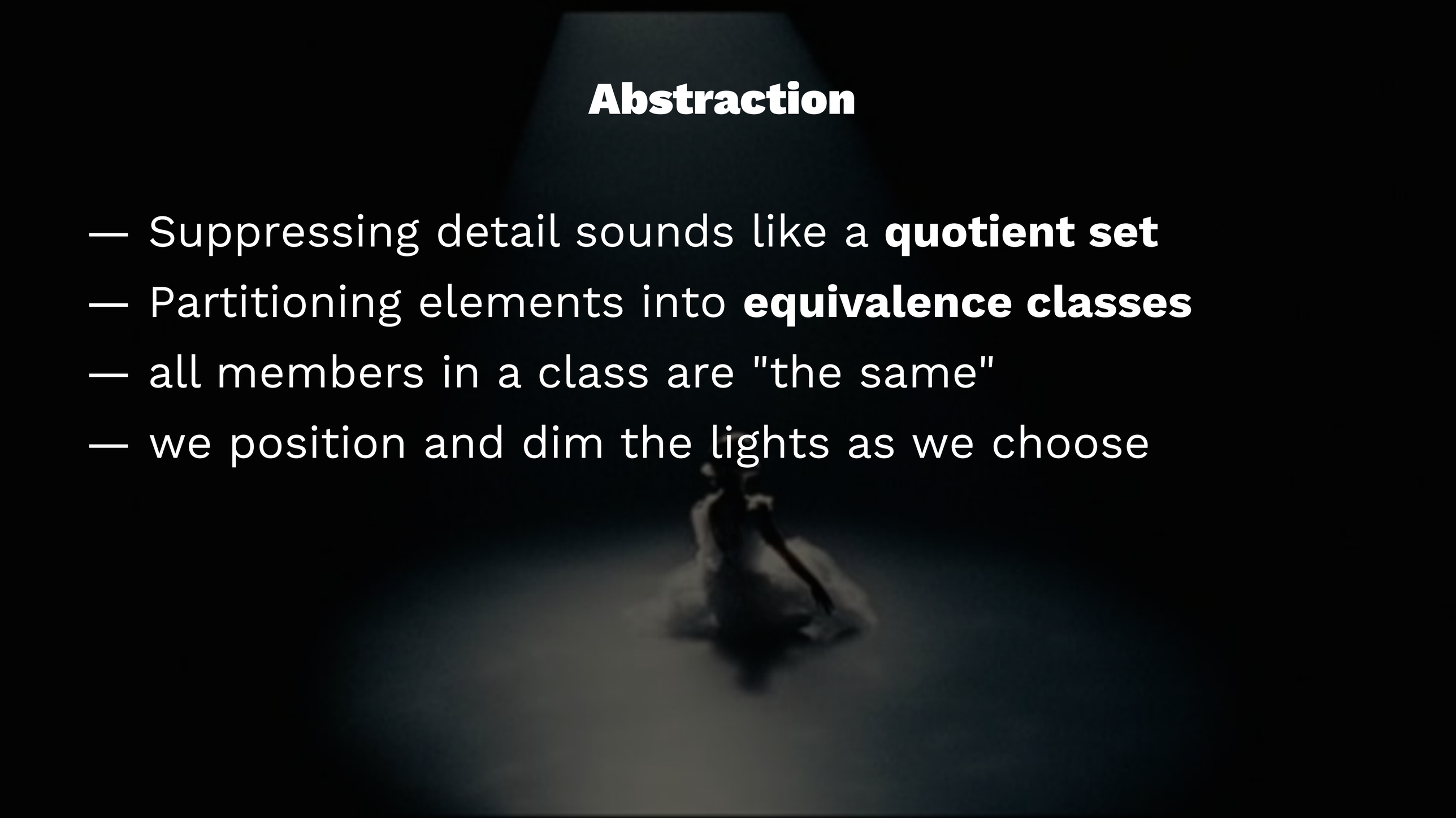
Generalization: It lays EGGS?

Abstraction, abstractly

— The suppression of irrelevant detail
— A concept with a "platonic" definition
— A **precise** "semantic level"
— Neither overspecified nor underspecified

Abstraction

— Suppressing detail sounds like a **quotient set**
— Partitioning elements into **equivalence classes**
— all members in a class are "the same"
— we position and dim the lights as we choose



Abstraction, concretely

— The set of natural numbers N
— An equivalence relation P = "congruent modulo 1"
— Equivalence classes: `{ 1, 3, ... }`, `{ 2, 4, ... }`
— These classes even have names

Abstraction

— I have objects B, C, and D

— Notice: B <: A, C <: A, D <: A

— A abstracts B, C, and D

Generalization

— I have property R which classifies B

— Notice: property S <: R classifies B, C, and D

— S generalizes R

— Note **contravariance**

Do these actually differ?

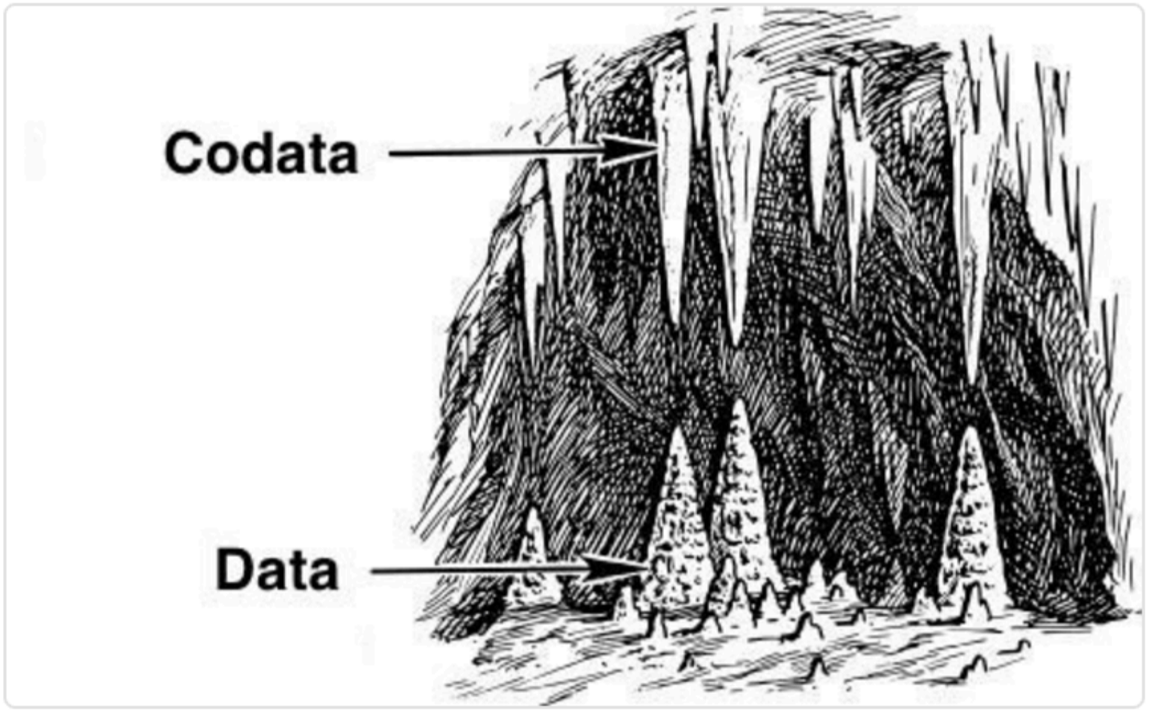
— The difference (in this conception) is the **arrow**
— This duality pervades not only math but life itself
— It is extension vs. intension

Extension	**<->**	**Intention**
abstract		generalize
unification		anti-unification
data		codata
being		doing
Set[A]		Indicator[A]
objects		attributes

Paul Phillips
@contrarivariant

Stalactite, from Greek stalaktos 'dripping'.
Stalagmite, from Greek stalagma 'a drop'.

Codata vs. Data.



The Marriage of Extent and Intent

What is this?

What am I describing?

- It has wings.	"A dragonfly?"
- It isn't an insect.	"A bat?"
- It lays eggs.	"A pterosaur?"
- It isn't a dinosaur.	"A platypus in a bird costume?"
- It flies.	"...in a hot air balloon?"
- It has feathers.	"A realistic avian drone?"

Imperfection

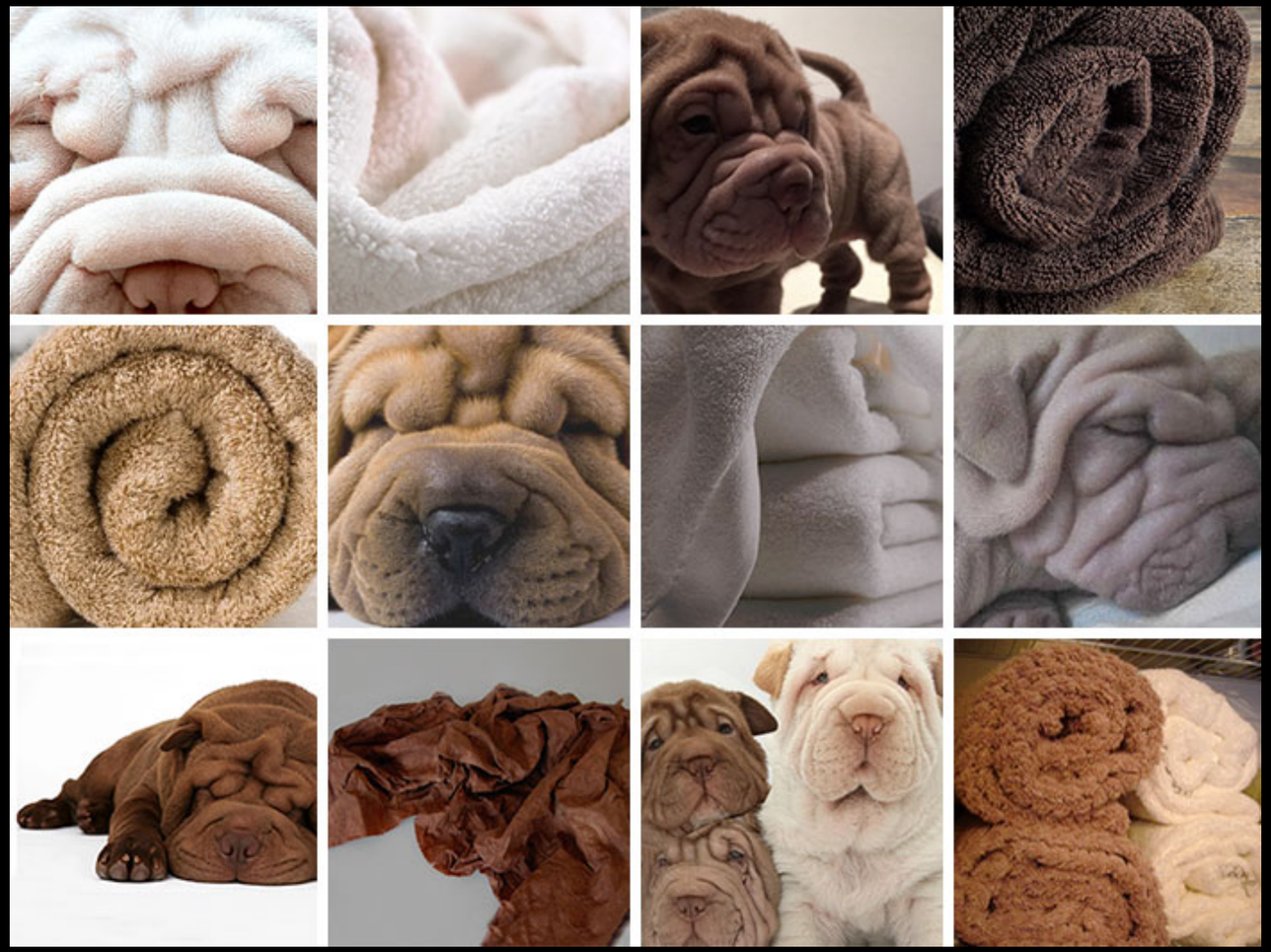
— We usually work with "objects" already in mind
— Naturally we will select properties they share
— So the flaws tend to be in the "negative space"
— How much "slack" is there in our abstraction?

Bagel abstraction

Mop abstraction

Muffin abstraction

Towel abstraction



Abstraction abstractly

— Ideally, extent and intent would converge
— An object set induces a common attribute set
— An attribute set induces a common object set
— When they correspond, we have something interesting
— This is the idea behind **formal concept analysis**

Formal Concept Analysis (FCA)

FCA finds practical application in fields including...

— data and text mining
— machine learning
— knowledge management
— the semantic web
— software development
— chemistry & biology

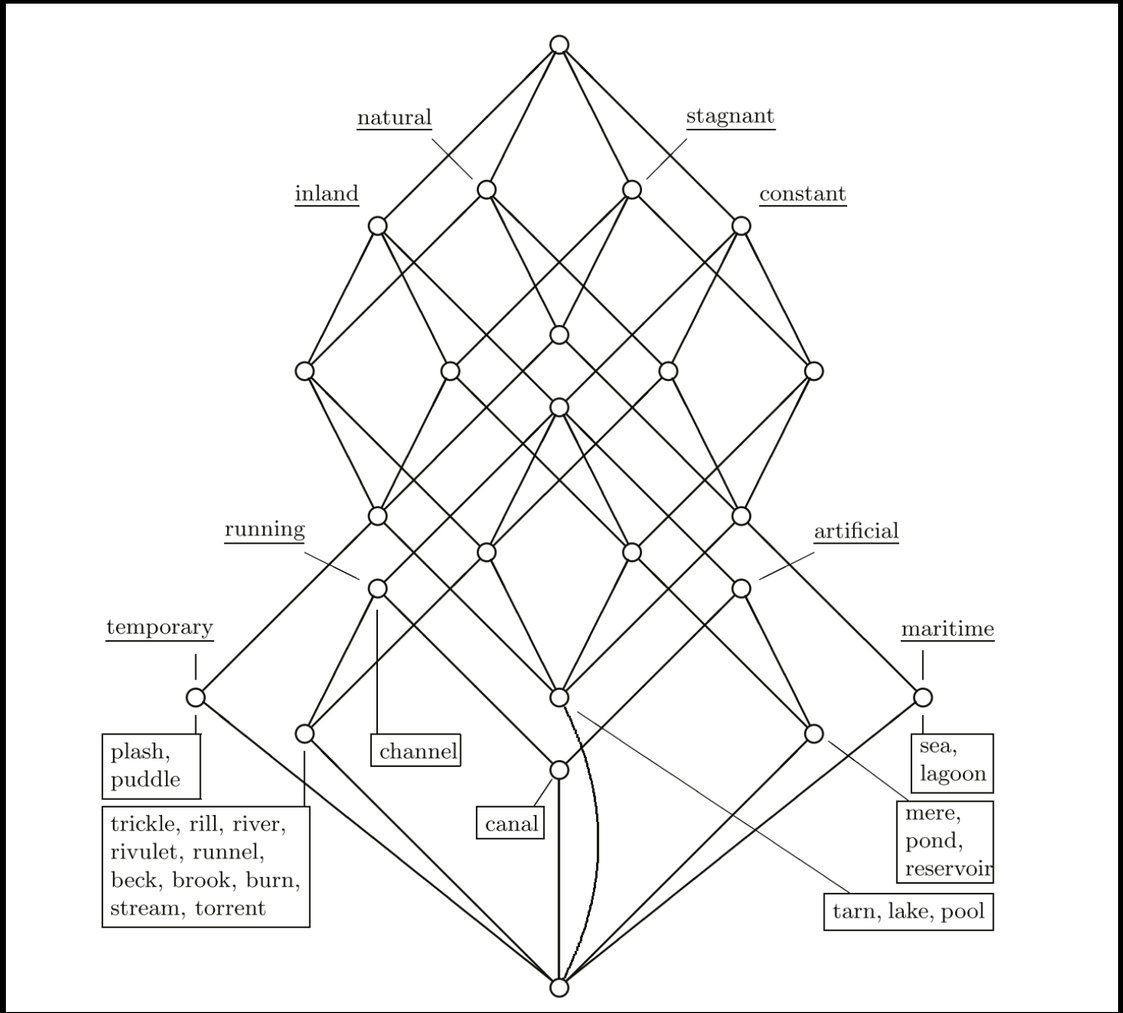
FCA (cont)

— Imagine facts as a matrix of booleans
— Each row is an **object**
— Each column is an **attribute**
— Each cell is true if object has attribute
— We can pre-filter as far as we like

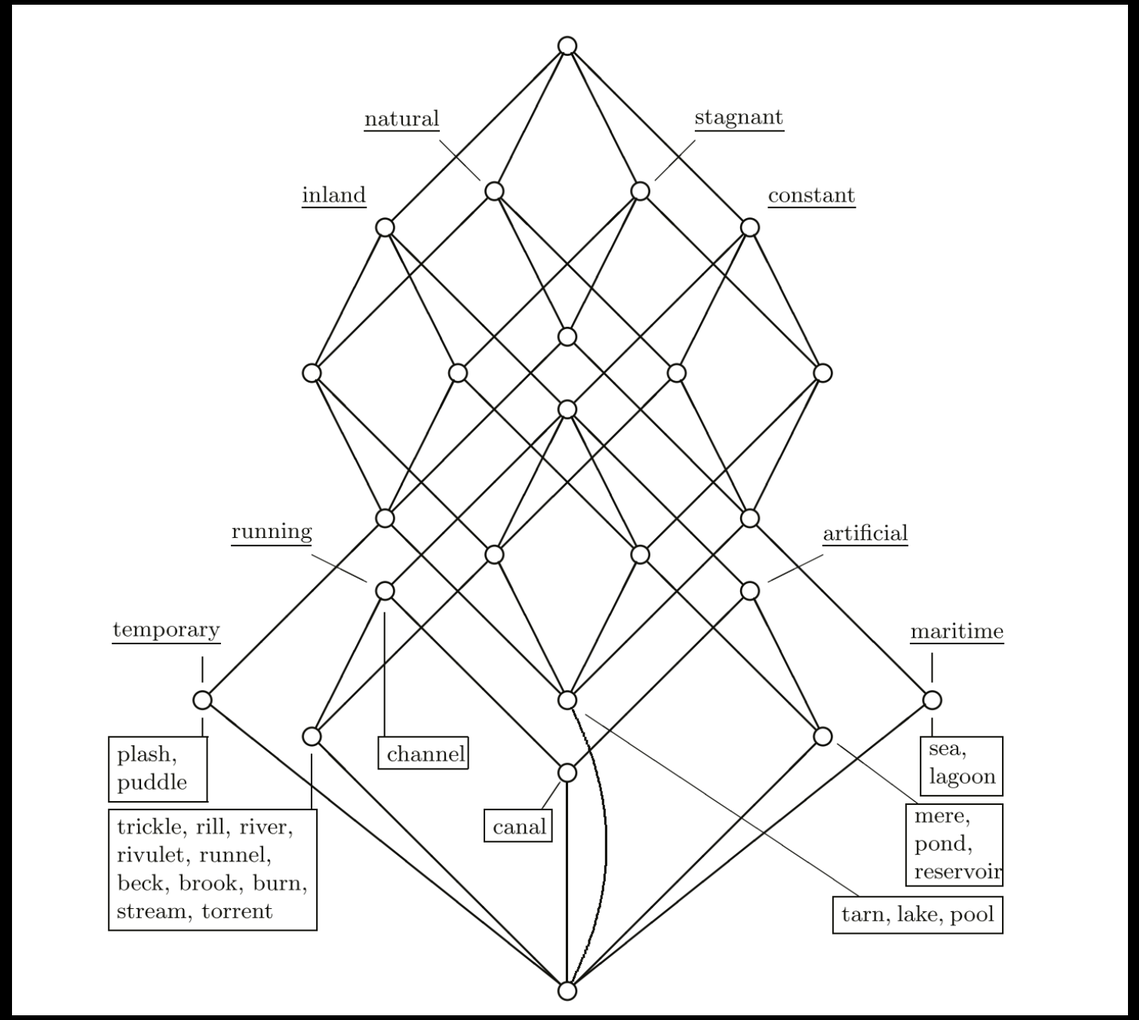
This is an incidence matrix

	natural	artificial	stagnant	running	inland	maritime	constant	temporary
tarn	X		X		X		X	
trickle	X			X	X		X	
rill	X			X	X		X	
beck	X			X	X		X	
rivulet	X			X	X		X	
runnel	X			X	X		X	
brook	X			X	X		X	
burn	X			X	X		X	
stream	X			X	X		X	
torrent	X			X	X		X	
river	X			X	X		X	
channel				X	X		X	
canal		X		X	X		X	
lagoon	X		X			X	X	
lake	X		X		X		X	
mere		X	X		X		X	
plash	X		X		X			X
pond		X	X		X		X	
pool	X		X		X		X	
puddle	X		X		X			X
reservoir		X	X		X		X	
sea	X		X			X	X	

This is its concept lattice



	natural	artificial	stagnant	running	inland	maritime	constant	temporary
tarn	X		X		X		X	
trickle	X			X	X		X	
rill	X			X	X		X	
beck	X			X	X		X	
rivulet	X			X	X		X	
runnel	X			X	X		X	
brook	X			X	X		X	
burn	X			X	X		X	
stream	X			X	X		X	
torrent	X			X	X		X	
river	X			X	X		X	
channel				X	X		X	
canal		X		X	X		X	
lagoon	X		X			X	X	
lake	X		X		X		X	
mere		X	X		X		X	
plash	X		X		X			X
pond		X	X		X		X	
pool	X		X		X		X	
puddle	X		X		X			X
reservoir		X	X		X		X	
sea	X		X			X	X	



Formal Concepts

— A formal context is K = (G, M, I) where
— G, a set of objects (Gegenstand = object)
— M, a set of attributes (Merkmal = characteristic)
— I, a binary relation GxM (Inzidenzrelation = incidence)
— From the formal context we derive the **concept lattice**

Formal Concepts (cont)

— For any set A ⊆ G and any set B ⊆ M
— The attribute set common to objects in A is A'
— The objects which have all attributes in B is B'
— If A=B' and A'=B, then (A,B) is a **formal concept**.

Formal Concepts (cont)

— The concept lattice orders these formal concepts
— The ordering is set inclusion
— (A, B) is a subconcept of (A′, B′) if $A \subseteq A'$ (and $B' \subseteq B$)
— Extent is covariant and intent is contravariant
— A subconcept has fewer objects and more attributes

Formal Concepts (cont)

— $I^{\uparrow}$: P(G) => P(M) (intersection of input attribute sets)

— $I^{\downarrow}$: P(M) => P(G) (set of objects with all attributes)

— Compose both ways to create closure operators:

— $I^{\uparrow}{}^{\downarrow}$: P(G) => P(G) and $I^{\downarrow}{}^{\uparrow}$: P(M) => P(M)

— Now for any set of objects A...

— ($I^{\uparrow}I^{\downarrow}(A)$, $I^{\uparrow}(A)$) is a formal concept

Formal Concepts (cont)

— Given formal concept (A,B)
— The objects in A are the concept's **extent**.
— The attributes in B are the concept's **intent**.
— There is a **Galois correspondence** between objects and attributes.

Darwin's 'survival of the fittest' is really a special case of a more general law of **survival of the stable**.

It may be a class of entities that come into existence at a sufficiently high rate to **deserve a collective name**.

An EXTENDED ladder of functional programming

Profunctor intuition

Jean Bénabou, who invented the term and originally used “profunctor,” now prefers “distributor,” which is supposed to carry the intuition that a distribut**or** generalizes a funct**or** in a similar way to how a distribution generalizes a function.
— n-lab

STANDARDIZED LADDER OF FUNCTIONAL PROGRAMMING

FUNCTIONAL PROGRAMMING FLAVOR: STATICALLY-TYPED, CATEGORY-THEORETIC — HASKELL, PURESCRIPT, SCALA, ETC

The LambdaConf Ladder of Functional Programming (LOFP) is a standardized grouping of concepts and skills relevant to functional programming in a statically-typed, category-theoretic programming environment. LOFP is used to categorize LambdaConf's workshops, talks, presentations, books, and courseware, so that aspiring functional programmers can better identify material that matches their backgrounds.

FIRE KERAMIK

CONCEPTS

- Immutable Data
- Second-Order Functions
- Constructing & Destructuring
- Function Composition
- First-Class Functions & Lambdas

SKILLS

- Use second-order functions (map, filter, fold) on immutable data structures
- Destructure values to access their components
- Use data types to represent optionality
- Read basic type signatures
- Pass lambdas to second-order functions

FIRE LUBLINE

CONCEPTS

- Algebraic Data Types
- Pattern Matching
- Parametric Polymorphism
- General Recursion
- Type Classes, Instances, & Laws
- Lower-Order Abstractions (Equal, Semigroup, Monoid, etc)
- Referential Transparency & Totality
- Higher-Order Functions
- Partial-Application, Currying, & Point-Free Style

SKILLS

- Solve problems without nulls, exceptions, or type casts
- Process & transform recursive data structures using recursion
- Able to use functional programming "in the small"
- Write basic monadic code for a concrete monad
- Create type class instances for custom data types
- Model a business domain with ADTs
- Write functions that take and return functions
- Reliably identify & isolate pure code from impure code
- Avoid introducing unnecessary lambdas & named parameters

ICE SKRIG

CONCEPTS

- Generalized Algebraic Data Types
- Higher-Kinded Types
- Rank-N Types
- Folds & Unfolds
- Higher-Order Abstractions (Category, Functor, Monad)
- Basic Optics
- Efficient Persistent Data Structures
- Existential Types
- Embedded DSLs using Combinators

SKILLS

- Able to use functional programming "in the large"
- Test code using generators and properties
- Write imperative code in a purely functional way through monads
- Use popular purely functional libraries to solve business problems
- Separate decision from effects
- Write a simple custom lawful monad
- Write production medium-sized projects
- Use lenses & prisms to manipulate data
- Simplify types by hiding irrelevant data with existentials

FIRE BRUSHE

CONCEPTS

- Codata
- (Co)Recursion Schemes
- Advanced Optics
- Dual Abstractions (Comonad)
- Monad Transformers
- Free Monads & Extensible Effects
- Functional Architecture
- Advanced Functors (Exponential, Profunctors, Contravariant)
- Embedded DSLs using GADTs, Finally Tagless
- Advanced Monads (Continuation, Logic)
- Type Families, Functional Dependencies

SKILLS

- Design a minimally-powerful monad transformer stack
- Write concurrent and streaming programs
- Use purely functional mocking in tests
- Use type classes to modularly model different effects
- Recognize type patterns & abstract over them
- Use functional libraries in novel ways
- Use optics to manipulate state
- Write custom lawful monad transformers
- Use free monads / extensible effects to separate concerns
- Encode invariants at the type level
- Effectively use FDs / type families to create safer code

ICE THERMAGON

CONCEPTS

- High-Performance
- Kind Polymorphism
- Generic Programming
- Type-Level Programming
- Dependent-Types, Singleton Types
- Category Theory
- Graph Reduction
- Higher-Order Abstract Syntax
- Compiler Design for Functional Languages
- Profunctor Optics

SKILLS

- Design a generic, lawful library with broad appeal
- Prove properties manually using equational reasoning
- Design & implement a new functional programming language
- Create novel abstractions with laws
- Write distributed systems with certain guarantees
- Use proof systems to formally prove properties of code
- Create libraries that do not permit invalid states
- Use dependent-typing to prove more properties at compile-time
- Understand deep relationships between different concepts
- Profile, debug, & optimize purely functional code with minimal sacrifices

FANTASYLAND

FIRE BRUSHE

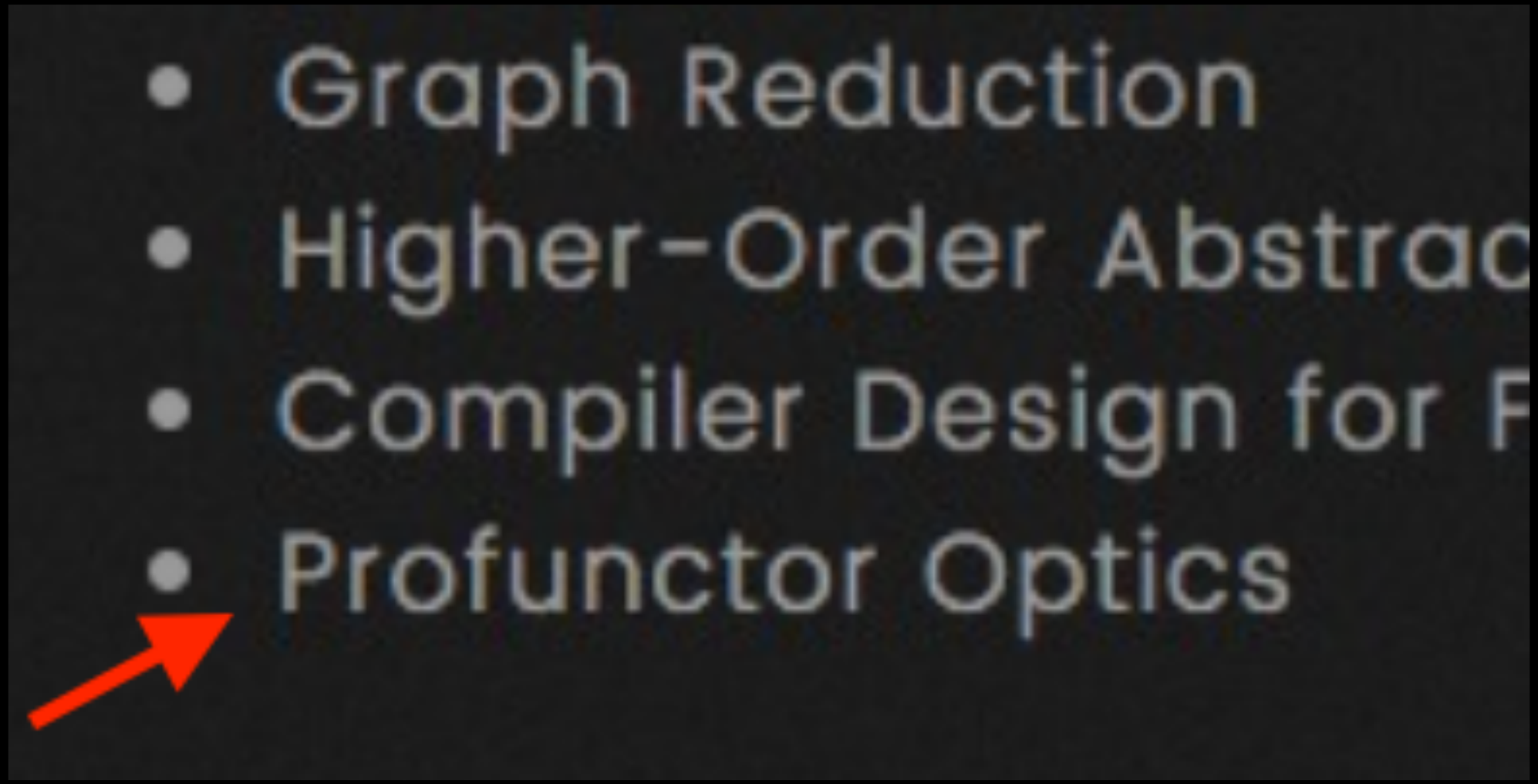
CONCEPTS

- Codata
- (Co)Recursion Schemes
- Advanced Optics
- Dual Abstractions (Comonad)
- Monad Transformers
- Free Monads & Extensible Effects
- Functional Architecture
- Advanced Functors (Exponential, Profunctors, Contravariant)
- Embedded DSLs using GADTs, Finally Tagless
- Advanced Monads (Continuation, Logic)
- Type Families, Functional Dependencies

ICE THERMAGON

CONCEPTS

- High-Performance
- Kind Polymorphism
- Generic Programming
- Type-Level Programming
- Dependent-Types, Singleton Types
- Category Theory
- Graph Reduction
- Higher-Order Abstract Syntax
- Compiler Design for Functional Languages
- Profunctor Optics


Graph Reduction
Higher-Order Abstrac
Compiler Design for F
Profunctor Optics

Several of you will have realised that the relation I can be viewed as a profunctor enriched in truth values, that the Galois connection is an adjunction between presheaf-type enriched categories and the concept lattice is the centre of the adjunction, i.e. the nucleus of the profunctor.


Welcome to the SECRET RUNG
"AXE-WIELDING FIRE MONSTER"
Profunctor Nuclei
NOTHING MORE TO LEARN