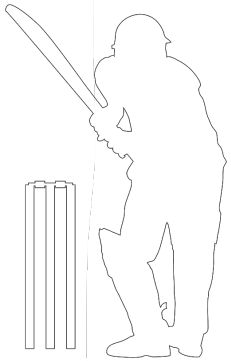

A Programmer's
Regret
Neglecting
Math at University
awalterschulze



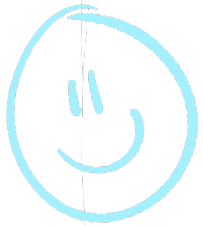
I don't like Cricket and I didn't like Math



TIME IN	TIME OUT	MINS	BATSMAN	RUNS AS SCORED	HOW OUT	BOWLER	TOTAL
11.00	5.23	323	1. M.J.SLATER		c GATTING	GOOCH	176 (244)
11.00	1.53	133	2. M.A.TAYLOR		RUN OUT		59 (110)
1.55	2.29	34	3. D.C.BOON		BOWLED	GOUGH	3 (24)
2.30	1.47	317	4. M.E.WAUGH		c STEWART	GOUGH	140 (215)
5.24	5.49	35	5. M.G.BEVAN		c HICK	GOUGH	7 (18)
5.50	11.24	34	6. S.K.WARNE		c RHODES	GOUGH	2 (25)
11.25	11.58	33	7. S.R.WAUGH		c HICK	DEFREITAS	19 (30)
12.01	12.34	33	8. I.A.HEALY		c HICK	DEFREITAS	7 (25)
12.36	12.54	18	9. C.J.McDERMOTT		c GOUGH	McGAGUE	2 (7)
12.56		20	10. T.B.A.MAY		NOT OUT		3 (21)
1.48	1.56	8	11. G.D.McGRATH		c GOUGH	McCAGUE	0 (5)

Math
What is it Good For?

A High Salary



A Tool for Specialised Problems



Every day Programming

A Tool for Specialised Problems

Games



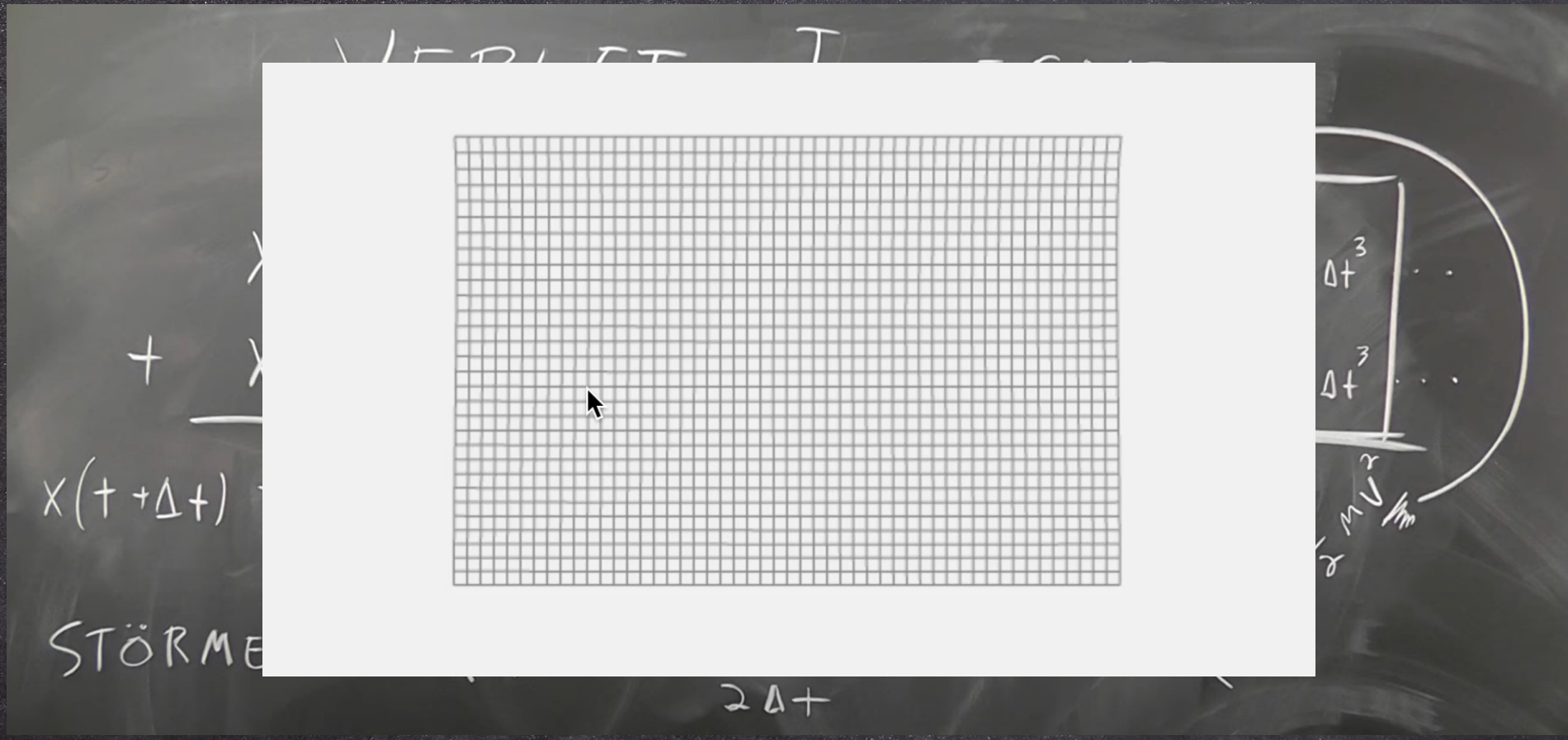
Search



AI



Games require Verlet Integration



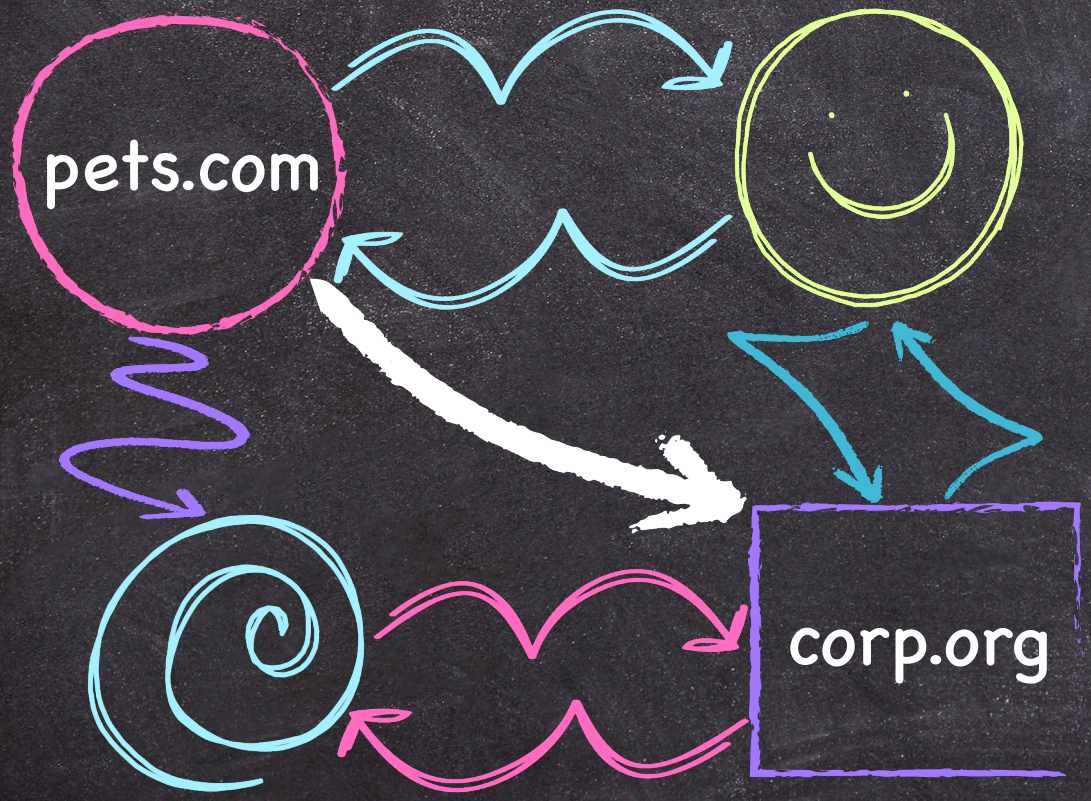
codepen.io/dissimulate/pen/nYQrNP

Leios Labs

TheHappieCat


SCORE 2406
Glumble - Exploring Ragdoll Physics with Goat Simulator

Search requires Linear Algebra



$$\begin{bmatrix} 0 & 0.5 & 0 & 0 \\ 0.33 & 0 & 0 & 0.5 \\ 0.33 & 0 & 0 & 0.5 \\ 0.33 & 0.5 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0.12 \\ 0.24 \\ 0.24 \\ 0.4 \end{bmatrix}$$

Digital Learning Hub - Imperial College London



This is the
Afrikaans
Coastguard.
What are you sighing about?

Programming is eating Math

Abstraction as an Appetiser

Composition as the main Cuisine

The Proof is in the Pudding

Abstraction as an Appetiser: Group

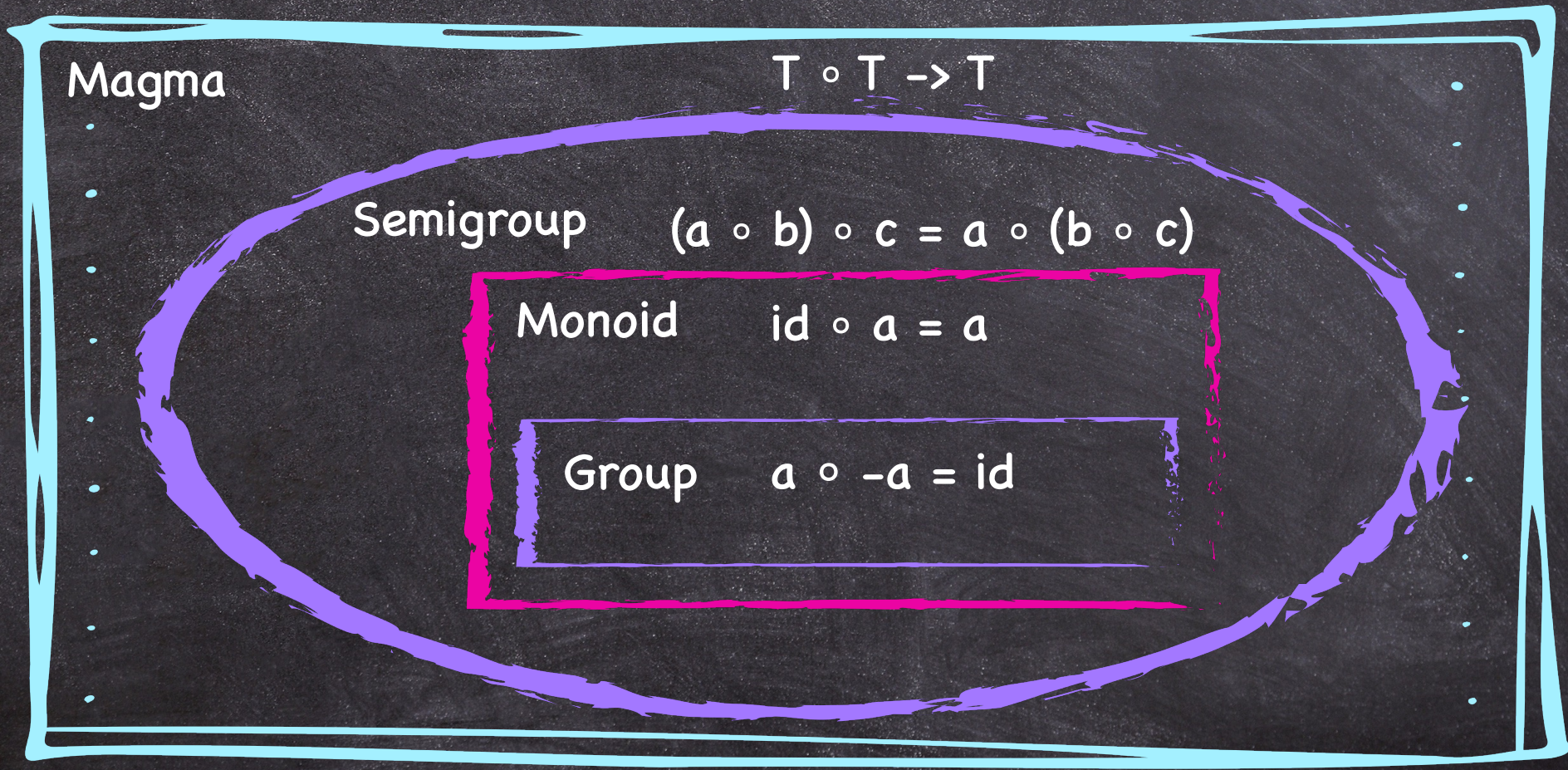
addition: +

- binary operator:
  T + T -> T
- identity:
  0 + a = a
- inverse: -
  a + (- a) = id
- associative:
  (a + b) + c =
  a + (b + c)

product: x

- binary operator:
  T x T -> T
- identity:
  1 x a = a
- inverse:
  a x (1 / a) = id
- associative:
  (a x b) x c =
  a x (b x c)

Abstraction as an Appetiser: Inheritance



Abstraction as an Appetiser: Inheritance

```
class Magma (α : Type u) where
  op : α → α → α

infixl:65 " ∘ " => Magma.op

class Semigroup (G : Type u) extends Magma G where
  is_assoc : ∀ a b c : G, (a ∘ b) ∘ c = a ∘ (b ∘ c)

class Monoid (M : Type u) extends Semigroup M where
  id: M
  left_identity: ∀ (a: M), id ∘ a = a
  right_identity: ∀ (a: M), a ∘ id = a
```

Abstraction as an Appetiser: Monoid

addition: +

- binary operator:
  T + T -> T
- identity:
  0 + a = a
- associative:
  (a + b) + c =
  a + (b + c)

Abstraction as an Appetiser: Monoids

Monoid	Identity	Operation
Addition over Integers	0	+
Multiplication over Integers	1	x
Concatenation over Strings	""	+
Extending over Lists	[]	extend
Union over Sets	{}	union
Any over Booleans	false	||
Overlay for Images	transparent	overlay image

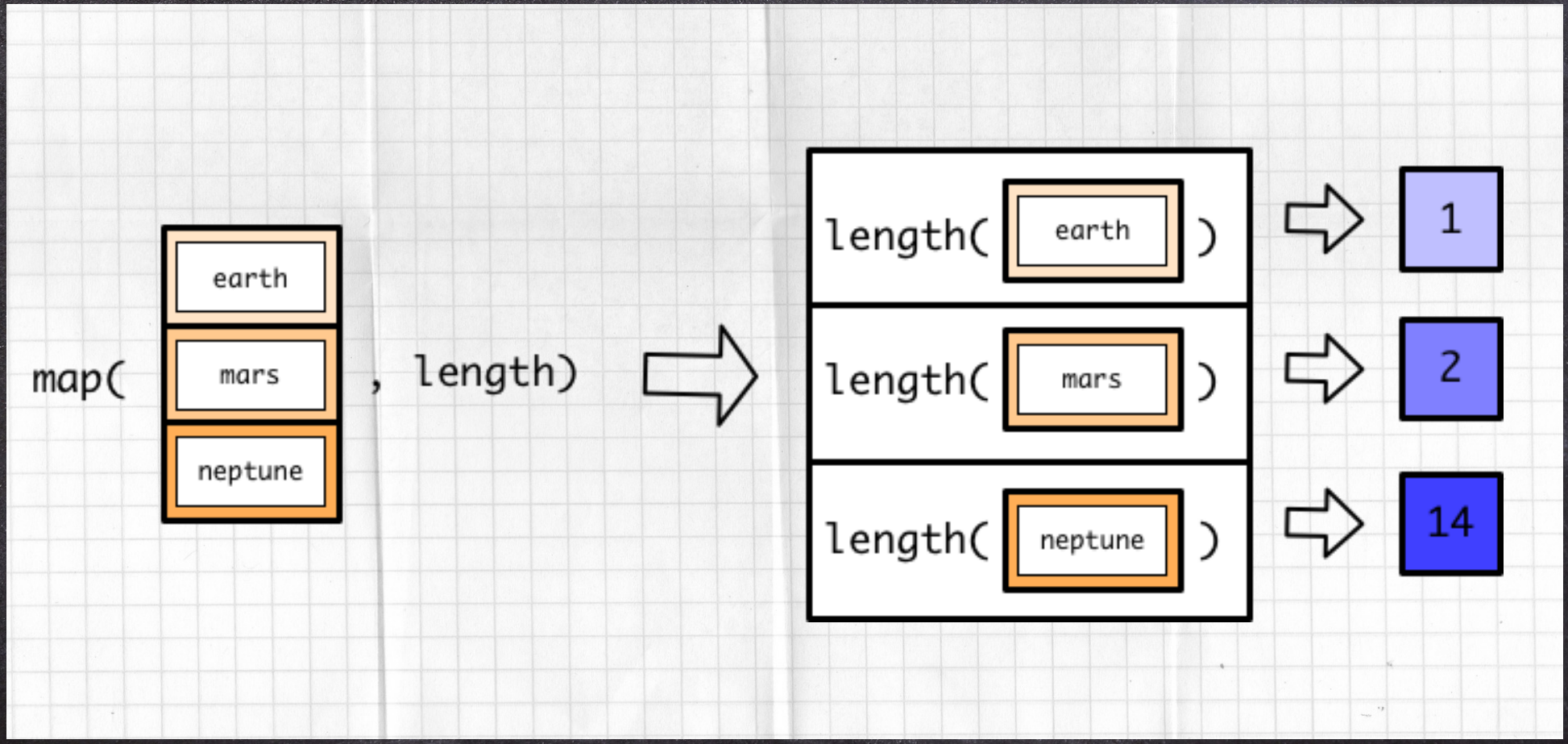
Abstraction => Common Code

```
def mconcat (m: Monoid α) (xs: List α): α := Id.run <| do
  let mut res: α := m.id
  for x in xs do
    res := res ∘ x
  return res

def any (xs: List Bool): Bool :=
  mconcat AnyMonoid xs

def all (xs: List Bool): Bool :=
  mconcat AllMonoid xs

def sum (xs: List Nat): Nat :=
  mconcat AddMonoid xs
```

Composition as the main Cuisine



Category Theory: Functors



dcl-prog.stanford.edu

Category Theory: Monads

Functor

Map: (f: a -> b) -> F a -> F b

Monads

Join : M (M a) -> M a

- Python List Comprehensions
  concat : list[list[a]] -> list[a]
- Rust Error Handling
- C# LINQ
- Scala Parser Combinators
- Haskell IO
- Erlang Actors
- Kotlin suspend
- Go context



awalterschulze - Monads for Go Programmers

Recursion Schemes

```haskell
nullable :: Regex -> Bool
nullable EmptySet          = False
nullable EmptyStr          = True
nullable (Char _)          = False
nullable (Concat a b)      = nullable a && nullable b
nullable (ZeroOrMore _)    = True
nullable (Or a b)          = nullable a || nullable b
```

Recursion Schemes

```
nullable :: Regex -> Bool
nullable EmptySet          = False
nullable EmptyStr          = True
nullable (Char _)          = False
nullable (Concat a b)      = nullable a && nullable b
nullable (ZeroOrMore _)    = True
nullable (Or a b)          = nullable a || nullable b
nullable (And a b)         = nullable a && nullable b
nullable (Interleave a b)  = nullable a && nullable b
```

Recursion Schemes

```haskell
nullFAlg :: FAlgebra RegexF Bool
nullFAlg EmptySet           = False
nullFAlg EmptyStr           = True
nullFAlg (Char _)           = False
nullFAlg (Concat a b)       = a && b
nullFAlg (ZeroOrMore _)     = True
nullFAlg (Or a b)           = a || b
nullFAlg (And a b)          = a && b
nullFAlg (Interleave a b) = a && b

nullable :: Regex -> Bool
nullable = cata nullFAlg
```

Algebraic Data Types: Game Show

Bool

```
type Bool = True | False
```

1 + 1 = 2

Algebraic Data Types: Game Show

Optional<Bool>

Optional<Bool> = None | Some Bool

1 + 2 = 3

Algebraic Data Types: Game Show

```
struct P {
  A: Bool
  B: Optional<Bool>
}
```

P{A: False, B: None}
P{A: False, B: Some False}
P{A: False, B: Some True}
P{A: True, B: None}
P{A: True, B: Some False}
P{A: True, B: Some True}

2 x 3 = 6

Algebraic Data Types: Game Show

```
struct P {
  A: Bool
  B: Optional<Bool>
  C: Bool
}
```

struct is a Product type

2 x 3 x 2 = 12

Algebraic Data Types: Sum Types

Elm

```
type Bool
  = True
  | False
```

```
type Sum p q
  = Left p
  | Right q
```

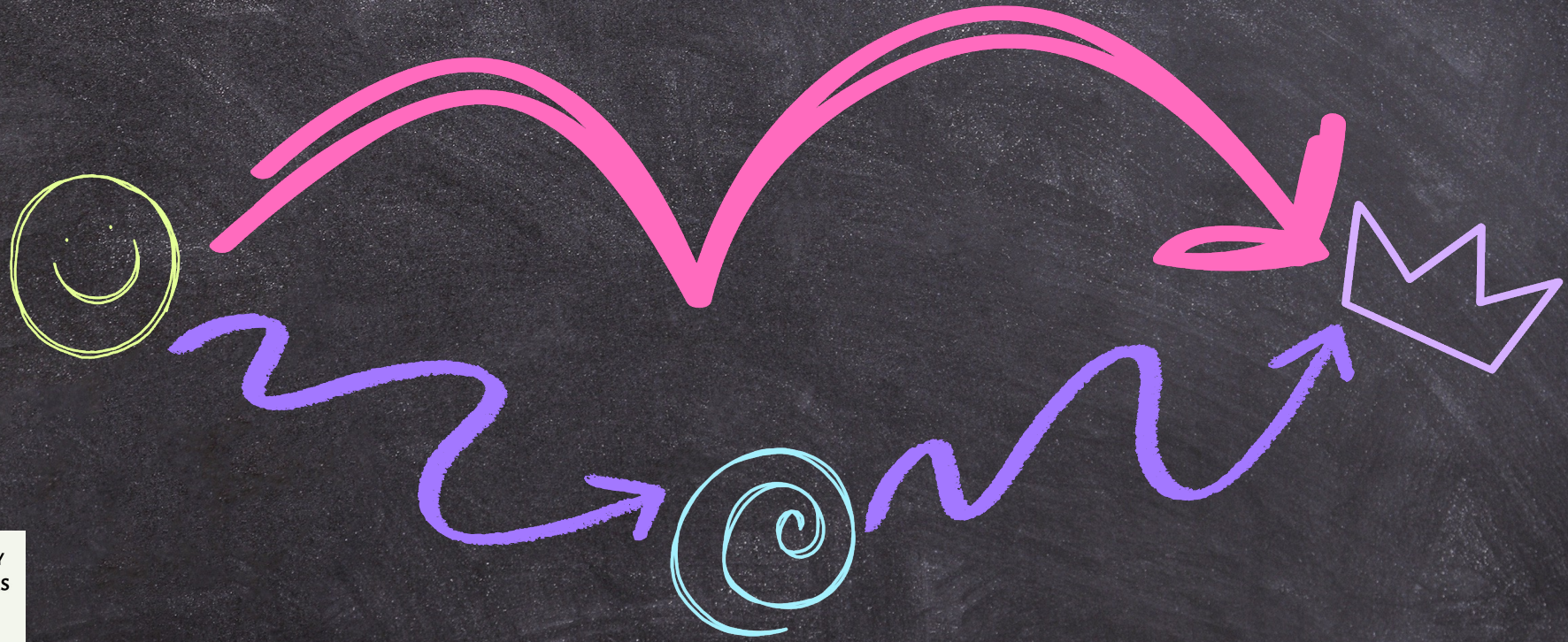
Rust

```
enum Sum<P, Q> {
  Left(P),
  Right(Q),
}
```

Kotlin

```
sealed class Sum<out P, out Q> {
  data class Left<P>(val data: P): Sum<P, Nothing>()
  data class Right<Q>(val error: Q): Sum<Nothing, Q>()
}
```

Category Theory for Composition





Category Theory for Programmers - Bartosz Milewski

The Proof is in the Pudding



Types <=> Propositions

Types

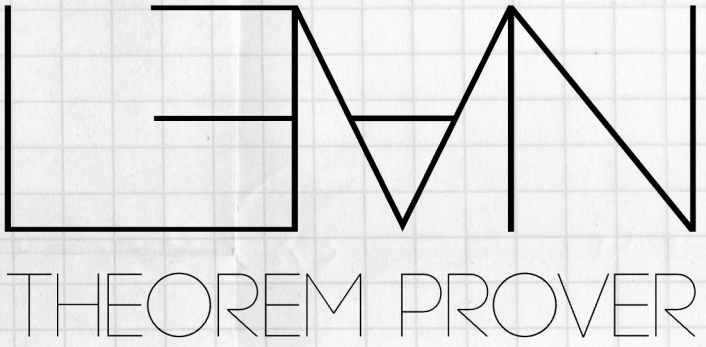
A × (B + C) -> (A × B) + (A × C)

```
distributive : Prod a (Sum b c) -> Sum (Prod a b) (Prod a c)
distributive {a, Left b} = Left (Prod a b)
distributive {a, Right c} = Right (Prod a c)
```

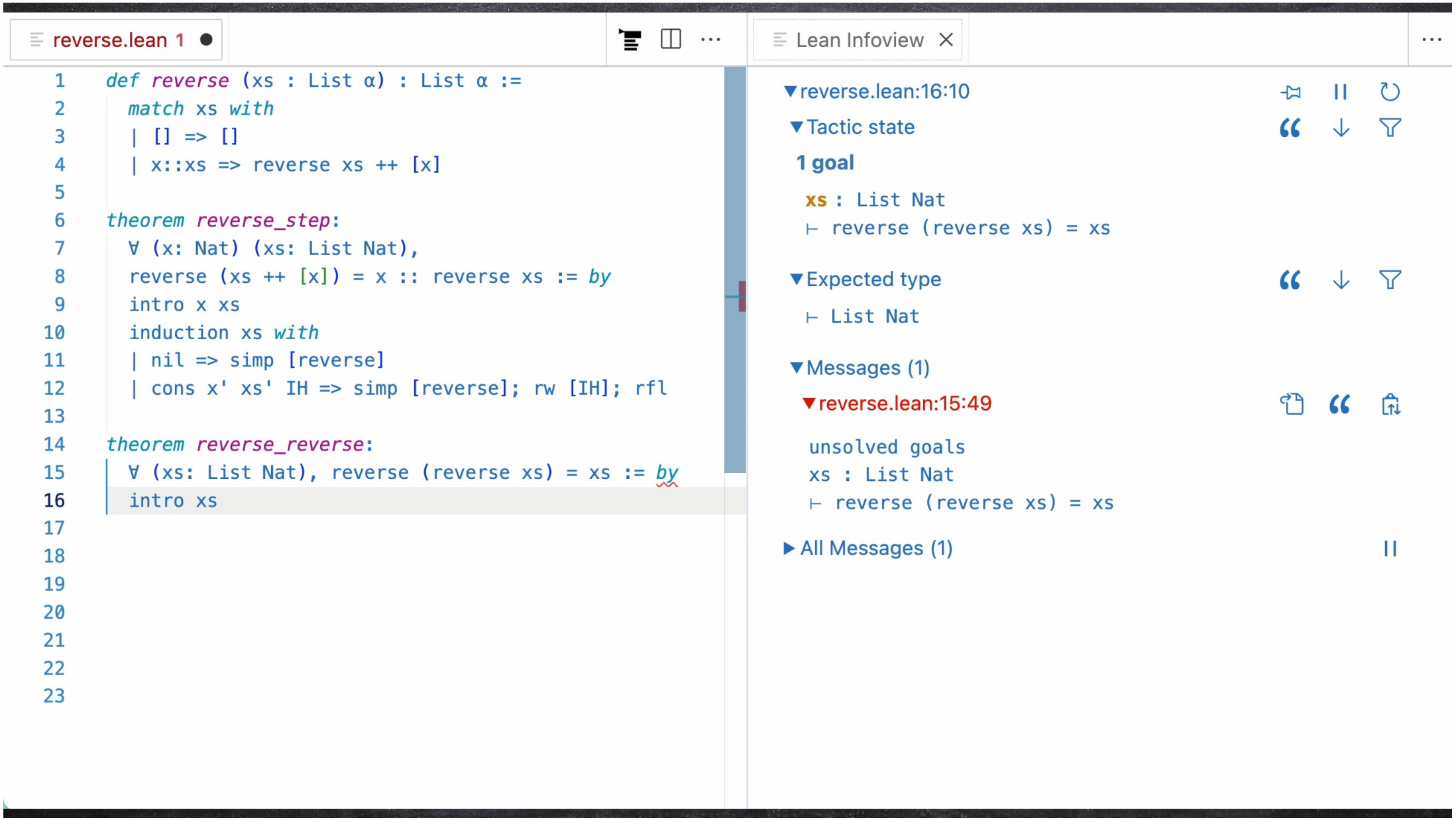
Propositions

A /\ (B \/ C) -> (A /\ B) \/ (A /\ C)

Programs <=> Proofs




```
def distributive_program (A B C: Type):
  A × (B ⊕ C) -> (A × B) ⊕ (A × C) :=
  fun h => match h.2 with
  | Sum.inl h2 => Sum.inl ⟨h.1, h2⟩
  | Sum.inr h2 => Sum.inr ⟨h.1, h2⟩

theorem distributive_proof (A B C: Prop):
  A /\ (B \/ C) -> (A /\ B) \/ (A /\ C) := by
  intro h; cases h.2 with
  | inl h2 => exact Or.inl ⟨h.1, h2⟩
  | inr h2 => exact Or.inr ⟨h.1, h2⟩
```


reverse.lean 1
Lean Infoview
1 def reverse (xs : List α) : List α :=
2 match xs with
3 | [] => []
4 | x::xs => reverse xs ++ [x]
5
6 theorem reverse_step:
7 ∀ (x: Nat) (xs: List Nat),
8 reverse (xs ++ [x]) = x :: reverse xs := by
9 intro x xs
10 induction xs with
11 | nil => simp [reverse]
12 | cons x' xs' IH => simp [reverse]; rw [IH]; rfl
13
14 theorem reverse_reverse:
15 ∀ (xs: List Nat), reverse (reverse xs) = xs := by
16 intro xs
17
18
19
20
21
22
23
reverse.lean:16:10
Tactic state
1 goal
xs : List Nat
⊢ reverse (reverse xs) = xs
Expected type
⊢ List Nat
Messages (1)
reverse.lean:15:49
unsolved goals
xs : List Nat
⊢ reverse (reverse xs) = xs
All Messages (1)

A taste of Coq … Proof Assistant



```
Theorem rev_rev A:
  forall (l : list A),
  rev (rev l) = l.
Proof.
induction l.
- reflexivity.
- simpl.
  rewrite rev_snoc_cons.
  rewrite IHl.
reflexivity.
Qed.
```

```
Lemma rev_snoc_cons A :
  forall (x : A) (l : list A),
  rev (l ++ [x]) = x :: rev l.
Proof.
induction l.
- reflexivity.
- simpl. rewrite IHl. simpl.
reflexivity.
Qed.
```

Proofs in Industry







Product	Verified
WebAssembly	Spec
Chrome and Firefox	Important parts of TLS
Zilliqa	Smart Contract Language
Rustbelt	Safe parts of "unsafe" Rust
CompCert	C compiler
Hedera	Proof of Stake Hashgraph
Cosette	SQL Optimizer
Tezos	Protocol









Correct Code by Construction by Walter Schulze & Paul Cadman

Logic Programming: Verse



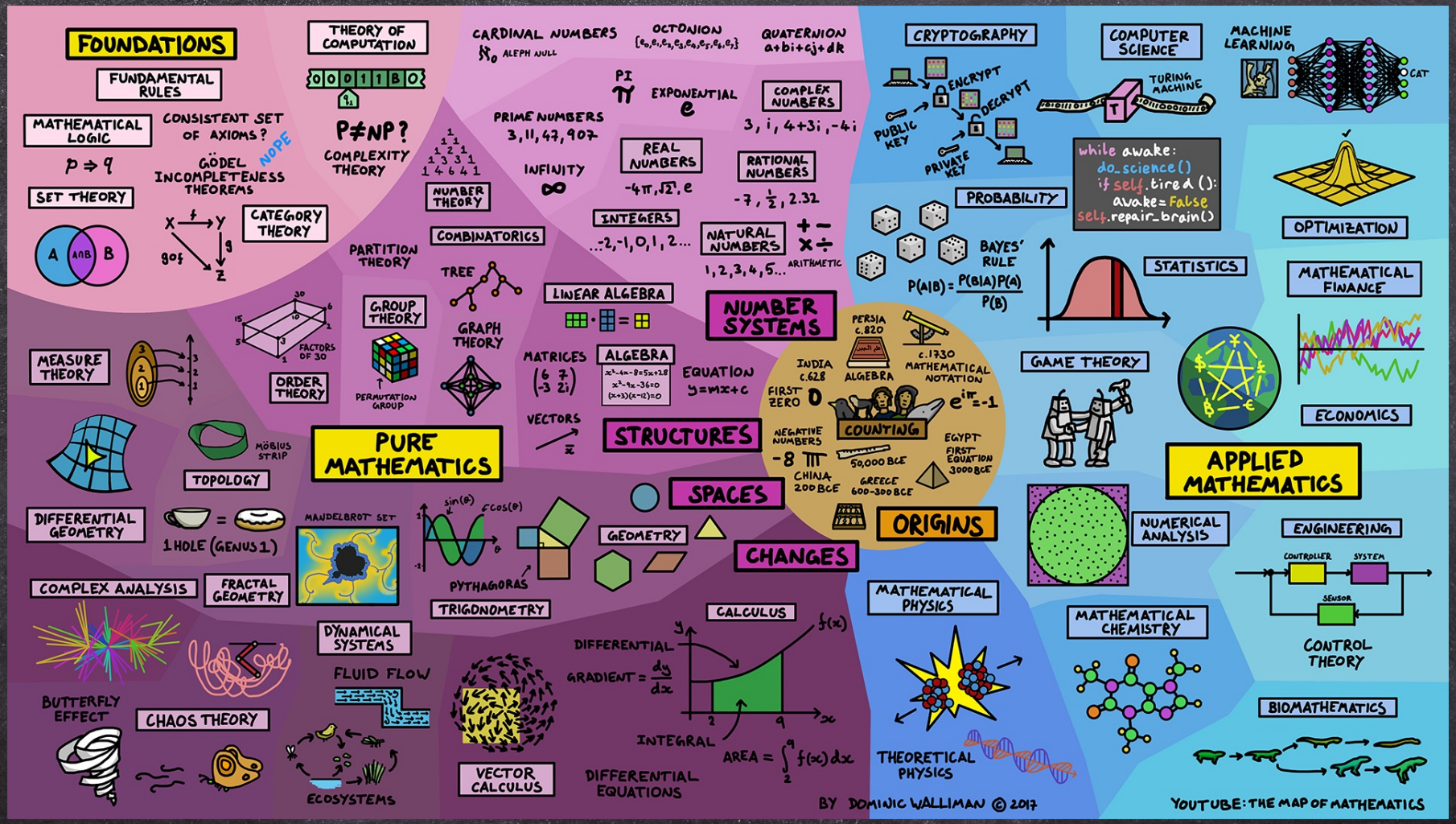
```
linkedList := class:
  Value: string
  Next: ?linkedList
```

$$
\begin{aligned}
&\forall\ xs\ ys\ zs \quad \in [string], \\
&\oplus\ xs\ ys\ zs \quad \equiv (xs = [] \land zs = ys) \\
&\qquad \lor (\exists\ head \in string, \\
&\qquad\quad \exists\ xtail\ ztail \in [string], \\
&\qquad\qquad xs = [head, xtail \ldots] \\
&\qquad\qquad \land\ zs = [head, ztail \ldots] \\
&\qquad\qquad \land \oplus\ xtail\ ys\ ztail\ )
\end{aligned}
$$

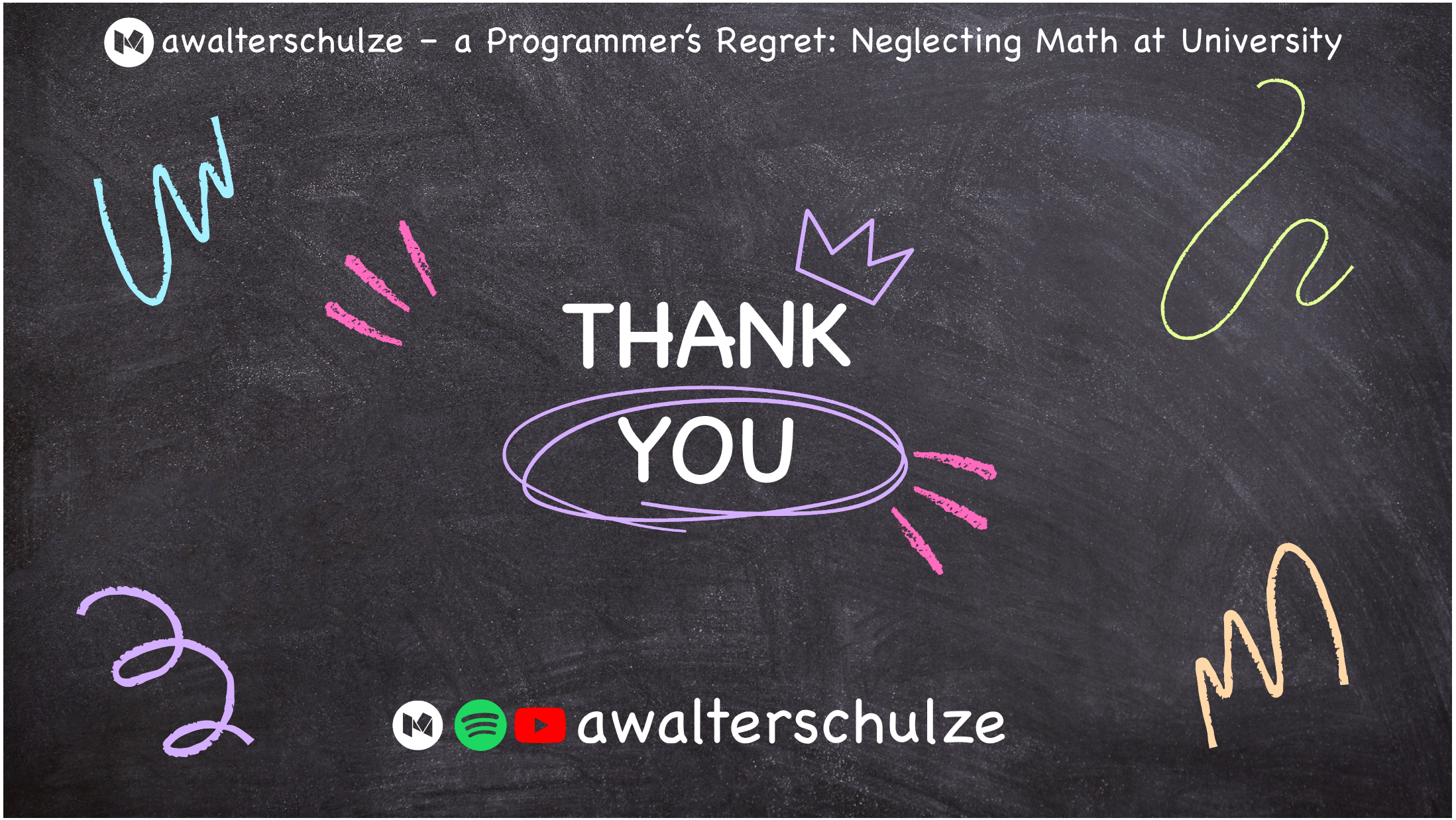
```
concato(xs, ys, zs: ?linkedList)<decides><transacts>: void :=
    ( xs = false; zs = ys)
  | ( head: string; xtail: ?linkedList; ztail: ?linkedList;
      xs = option{linkedList{Value := head, Next := xtail}};
      zs = option{linkedList{Value := head, Next := ztail}};
      concato(xtail, ys, ztail))
```

Logic Programming in Go - Walter Schulze

Map of Mathematics



dominicwalliman


awalterschulze – a Programmer's Regret: Neglecting Math at University
THANK
YOU
awalterschulze