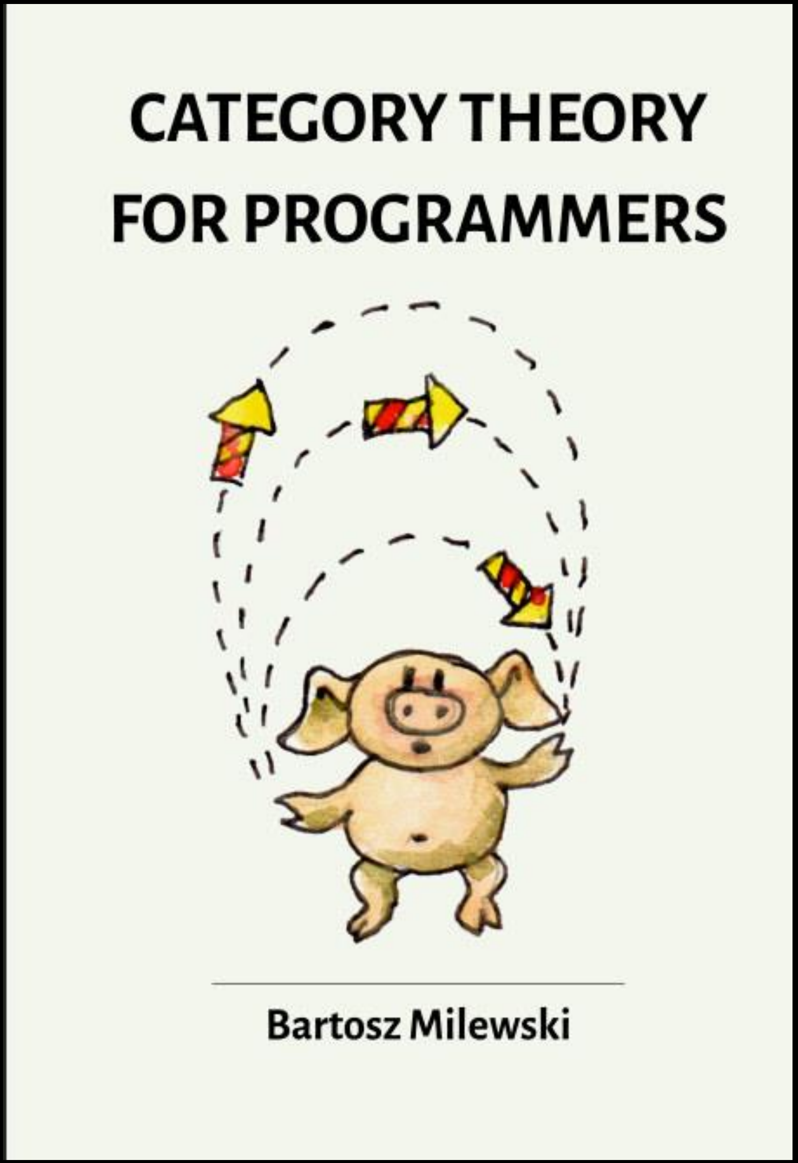
Java

Meetup

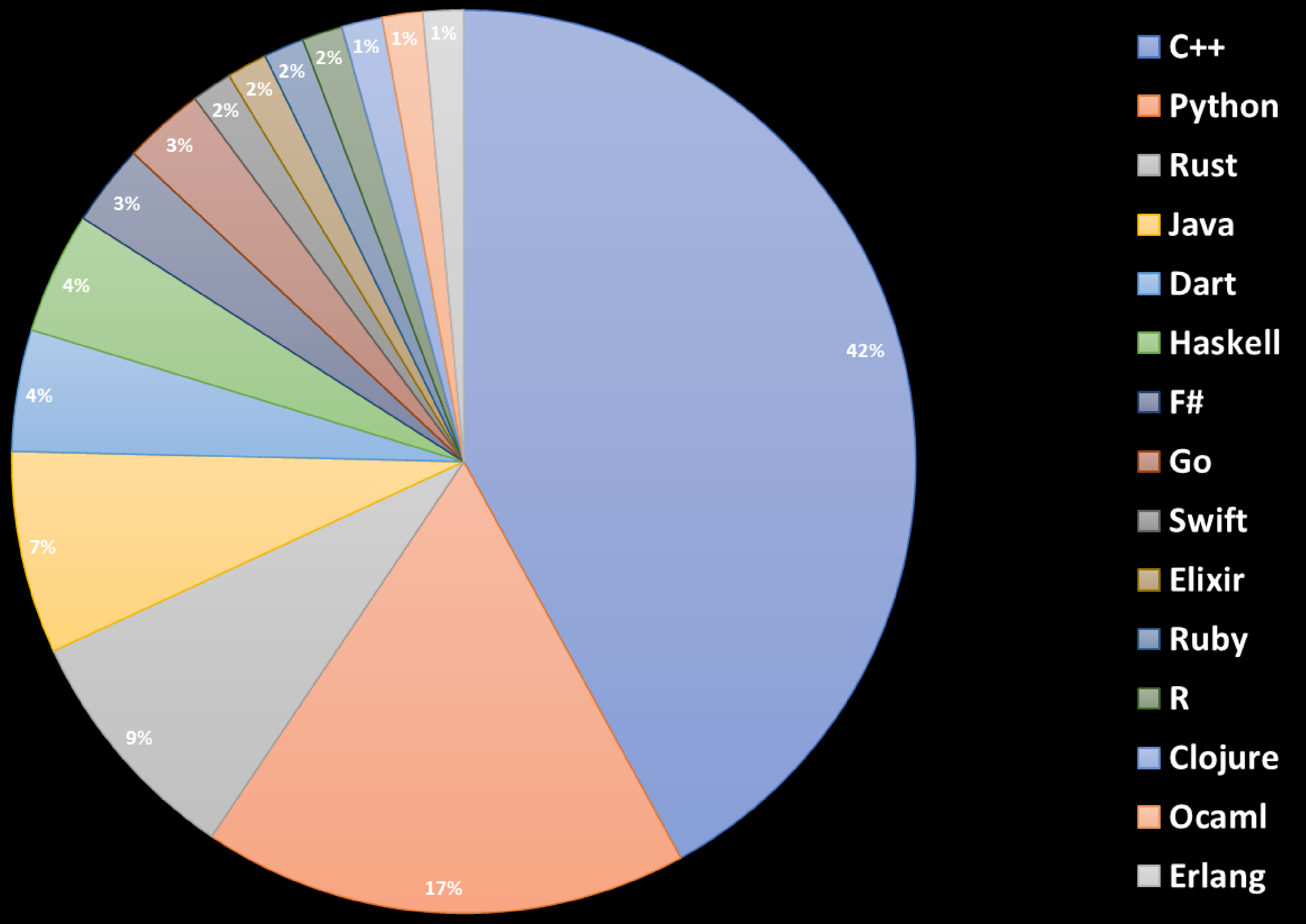
Friendly Environment Policy

Berlin Code of Conduct



Category Theory for Programmers

Chapter 2: Types & Functions


42%
17%
9%
7%
4%
4%
3%
3%
2%
2%
2%
2%
1%
1%
1%
C++
Python
Rust
Java
Dart
Haskell
F#
Go
Swift
Elixir
Ruby
R
Clojure
Ocaml
Erlang

THE CATEGORY OF TYPES AND FUNCTIONS plays an important role in programming, so let's talk about what types are and why we need them.

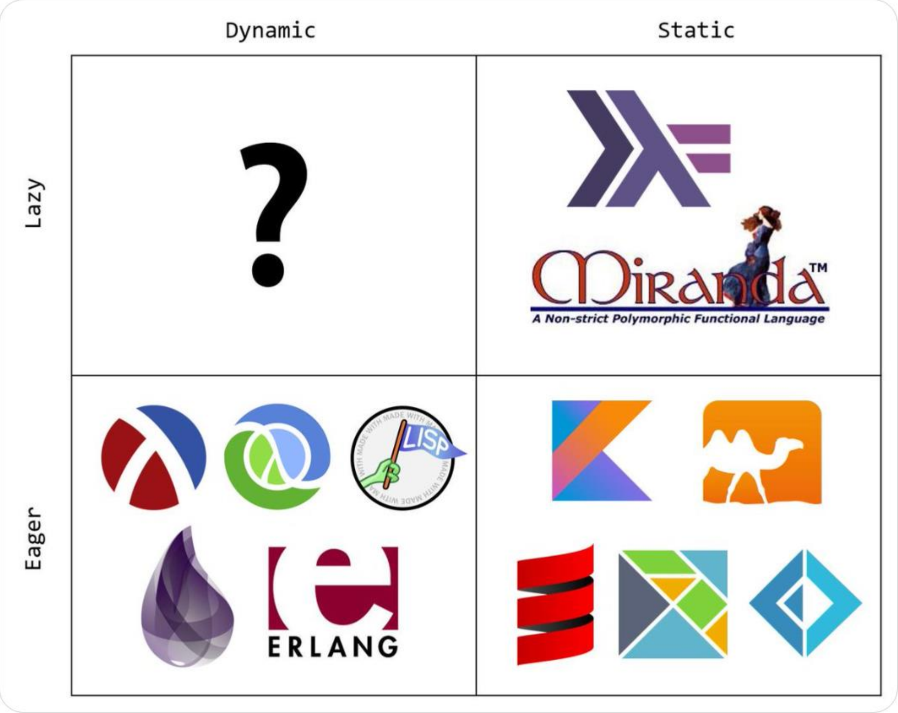
2.1 Who Needs Types?

There seems to be some controversy about the advantages of static vs. dynamic and strong vs. weak typing.

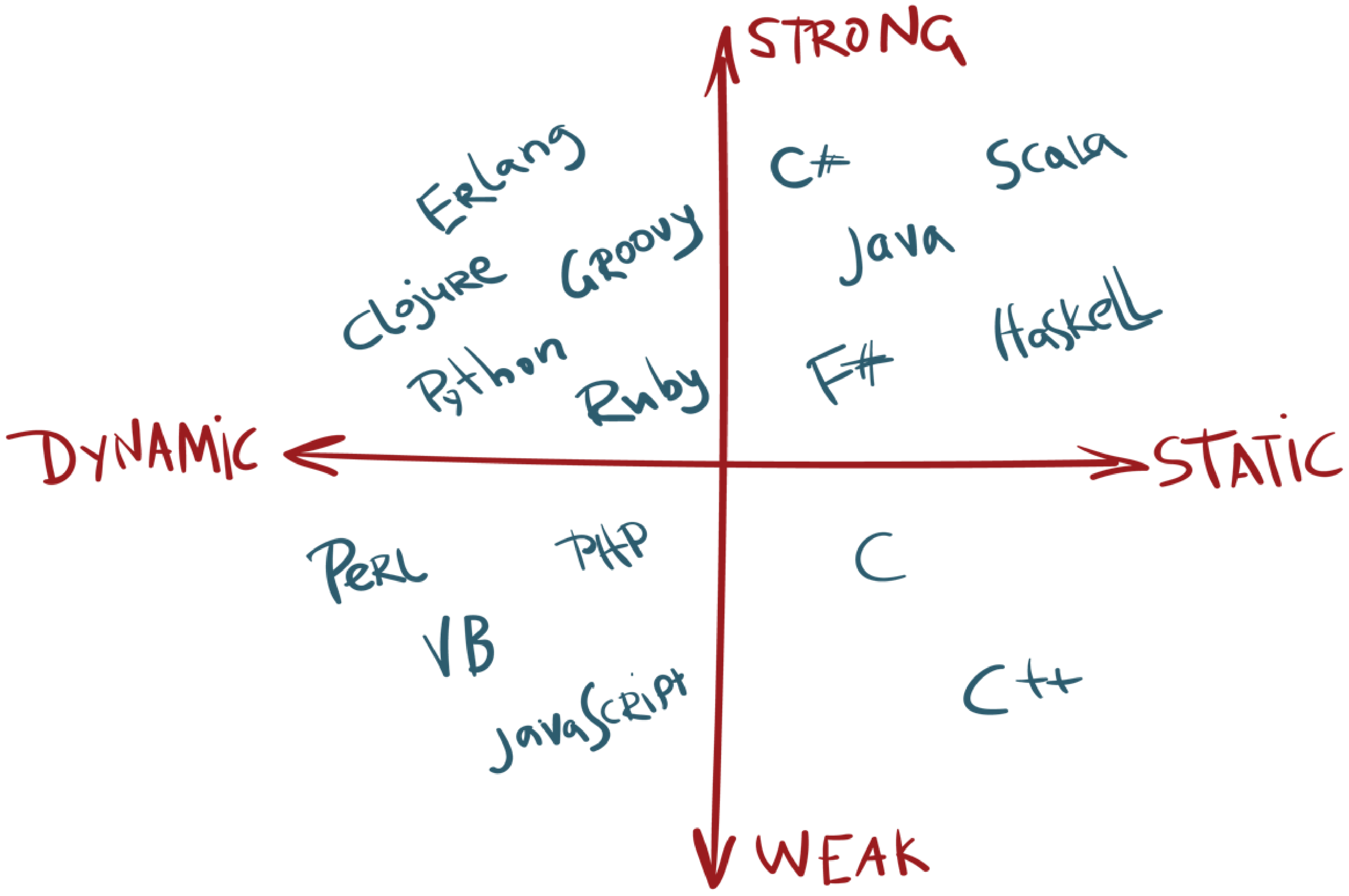
Conor Hoekstra
@code_report

Anyone know any lazy-by-default, dynamically typed functional programming languages? #functionalprogramming

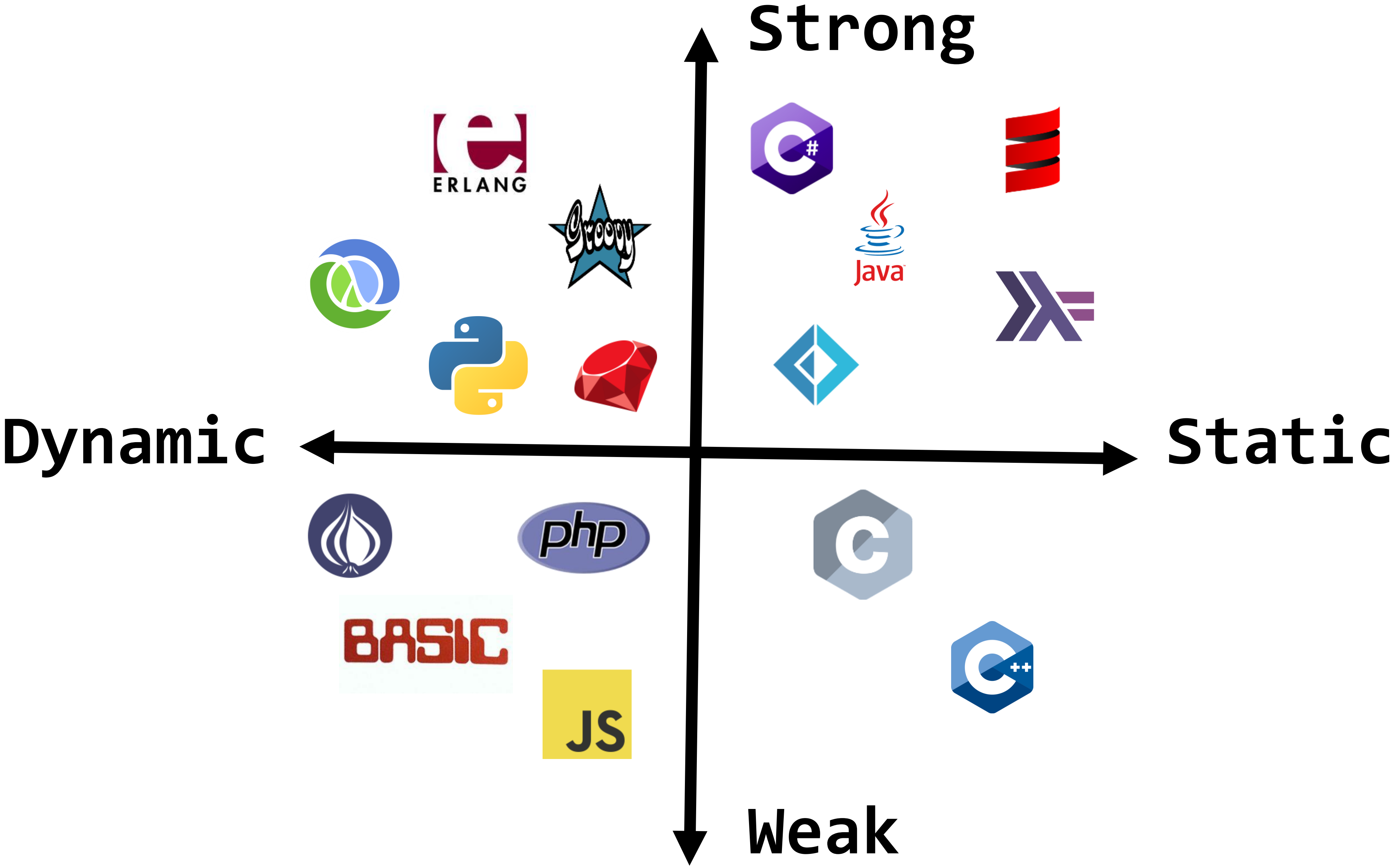
@planetclojure @kotlin @fsharporg @erlang_org @elixirlang @scala_lang @elmlang @OCamlLang @racketlang #haskell #lisp #schemelang #fsharp #clojure #scala #elixir #kotlin



3:50 AM · Aug 17, 2019 · Twitter Web App



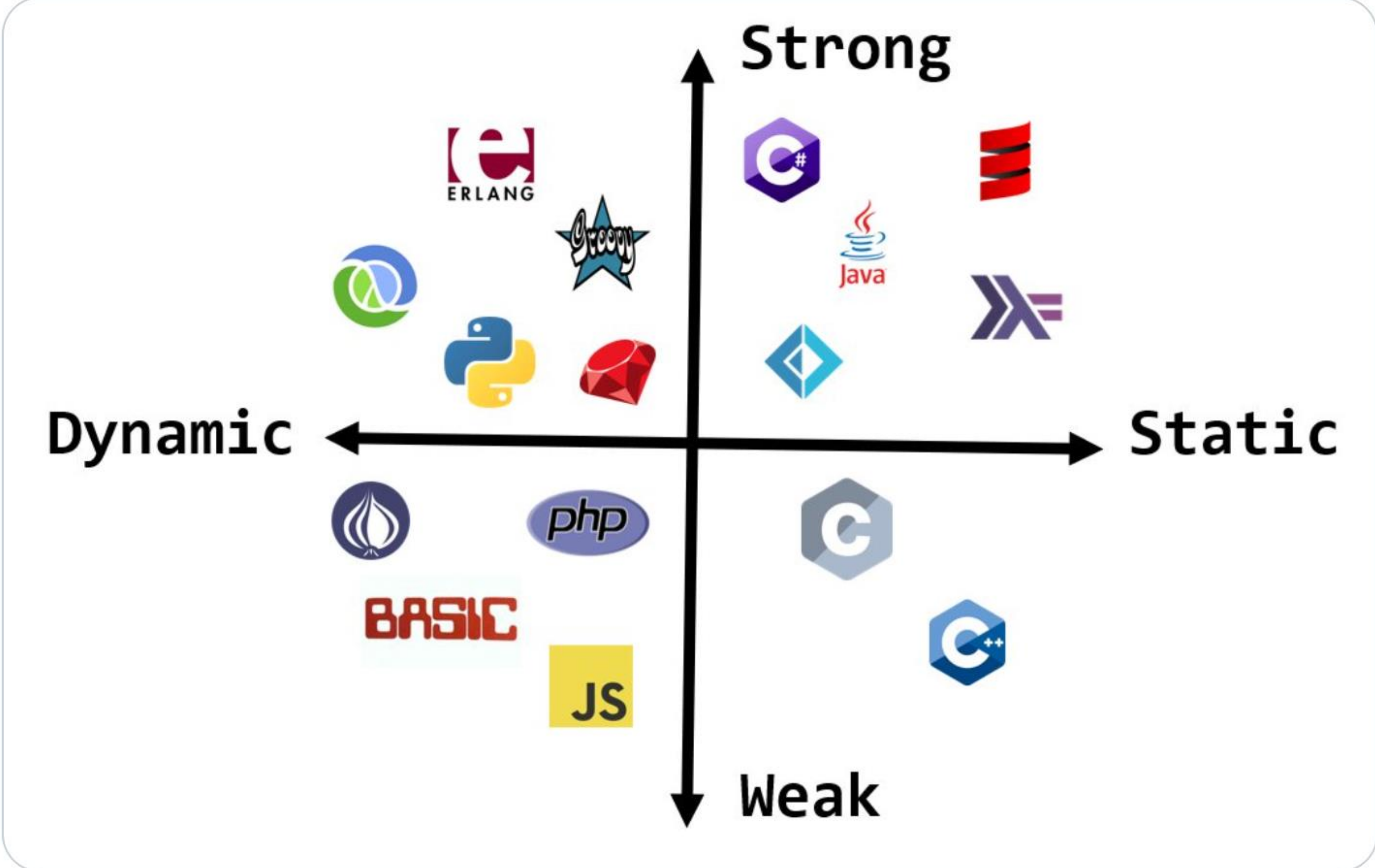
https://android.jlelse.eu/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b


Strong
Dynamic
Static
Weak
ERLANG
Groovy
Java
php
BASIC
JS

Cunningham's Law states "the best way to get the right answer on the internet is not to ask a question; it's to post the wrong answer."

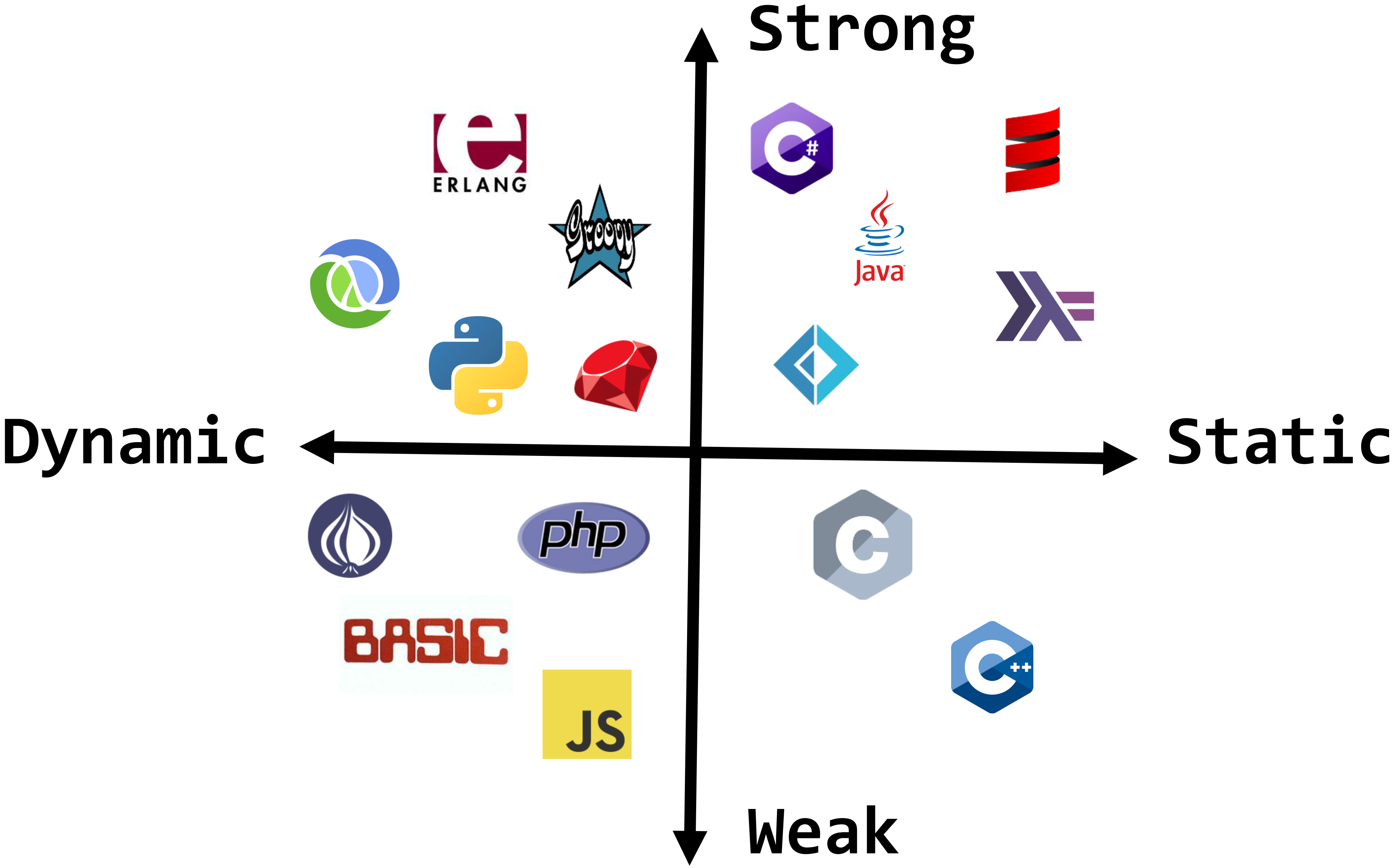
Programming Languages Virtual Meetup
@plvirtualmeetup

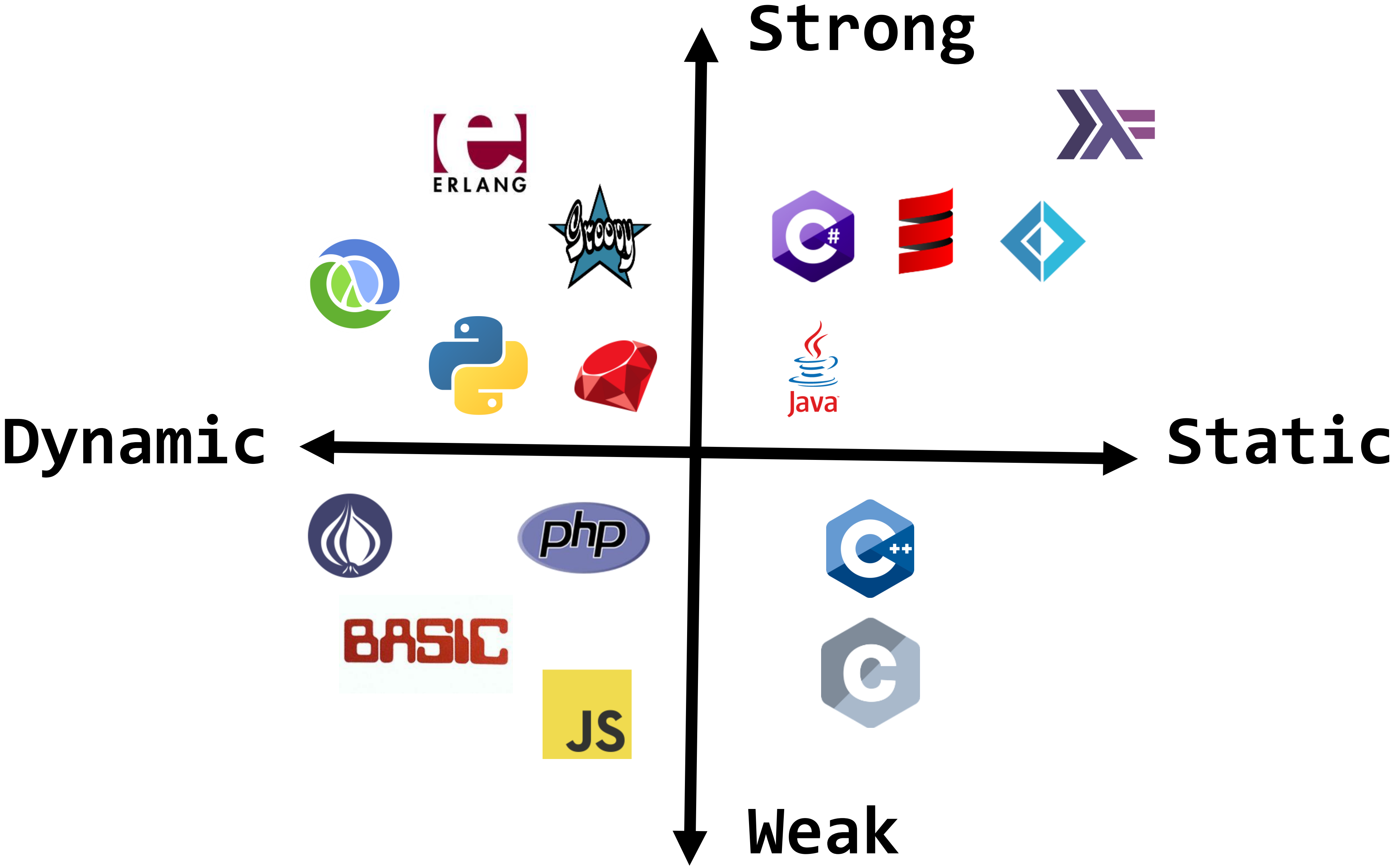
What #programminglanguages are missing and where do they go?



5:42 PM · Feb 5, 2021 · Twitter Web App

4 Retweets 1 Quote Tweet 19 Likes


Strong
ERLANG
Groovy
Java
Dynamic
Static
php
BASIC
JS
Weak


Strong
ERLANG
Groovy
Java
Dynamic
Static
php
BASIC
JS
Weak

2.3 What Are Types?

The simplest intuition for types is that they are sets of values. The type `Bool` (remember, concrete types start with a capital letter in Haskell) is a two-element set of `True` and `False`. Type `Char` is a set of all Unicode characters like `a` or `ą`.

But there is an alternative. It's called *denotational semantics* and it's based on math. In denotational semantics every programming construct is given its mathematical interpretation. With that, if you want to prove a property of a program, you just prove a mathematical theorem.

```haskell
fact n = product [1 .. n]
```

```cpp
int fact ( int n) {
    int result = 1;
    for (int i = 2; i <= n; ++ i)
        result *= i;
    return result;
}
```

```cpp
using namespace std::ranges;

auto fact(int n) -> int {
    auto vals = views::iota(1, n);
    return std::reduce(begin(vals), end(vals), 1, std::multiplies{});
}
```

```cpp
using namespace std::ranges;

auto fact(int n) -> int {
    return views::iota(1, n) | fold(1, std::multiplies{});
}
```

```rust
fn fact(n: i32) -> i32 {
    (1..=n).product()
}
```

```haskell
fact n = product [1 .. n]
```

```cpp
auto fact(int n) -> int {
    auto vals = views::iota(1, n);
    return std::reduce(begin(vals), end(vals), 1, std::multiplies{});
}
```

```rust
fn fact(n: i32) -> i32 {
    (1..=n).product()
}
```

Conor Hoekstra
@code_report

"I've been referring to it [Rust] 🦀 as a love child between Haskell and C++."

- quote from @roeschinc on Episode 77 of @fngeekery
Another awesome episode! #Haskell #cplusplus @rustlang

12:45 PM · Aug 26, 2019 from Sunnyvale, CA · Twitter for Android

In programming languages, functions that always produce the same result given the same input and have no side effects are called *pure functions*. In a pure functional language like Haskell all functions are pure.

Functions that can be implemented with the same formula for any type are called parametrically polymorphic.

Functions to `Bool` are called *predicates*. For instance, the Haskell library `Data.Char` is full of predicates like `isAlpha` or `isDigit`.

1. Define a higher-order function (or a function object) `memoize` in your favorite language. This function takes a pure function `f` as an argument and returns a function that behaves almost exactly like `f`, except that it only calls the original function once for every argument, stores the result internally, and subsequently returns this stored result every time it's called with the same argument. You can tell the memoized function from the original by watching its performance. For instance, try to memoize a function that takes a long time to evaluate. You'll have to wait for the result the first time you call it, but on subsequent calls, with the same argument, you should get the result immediately.

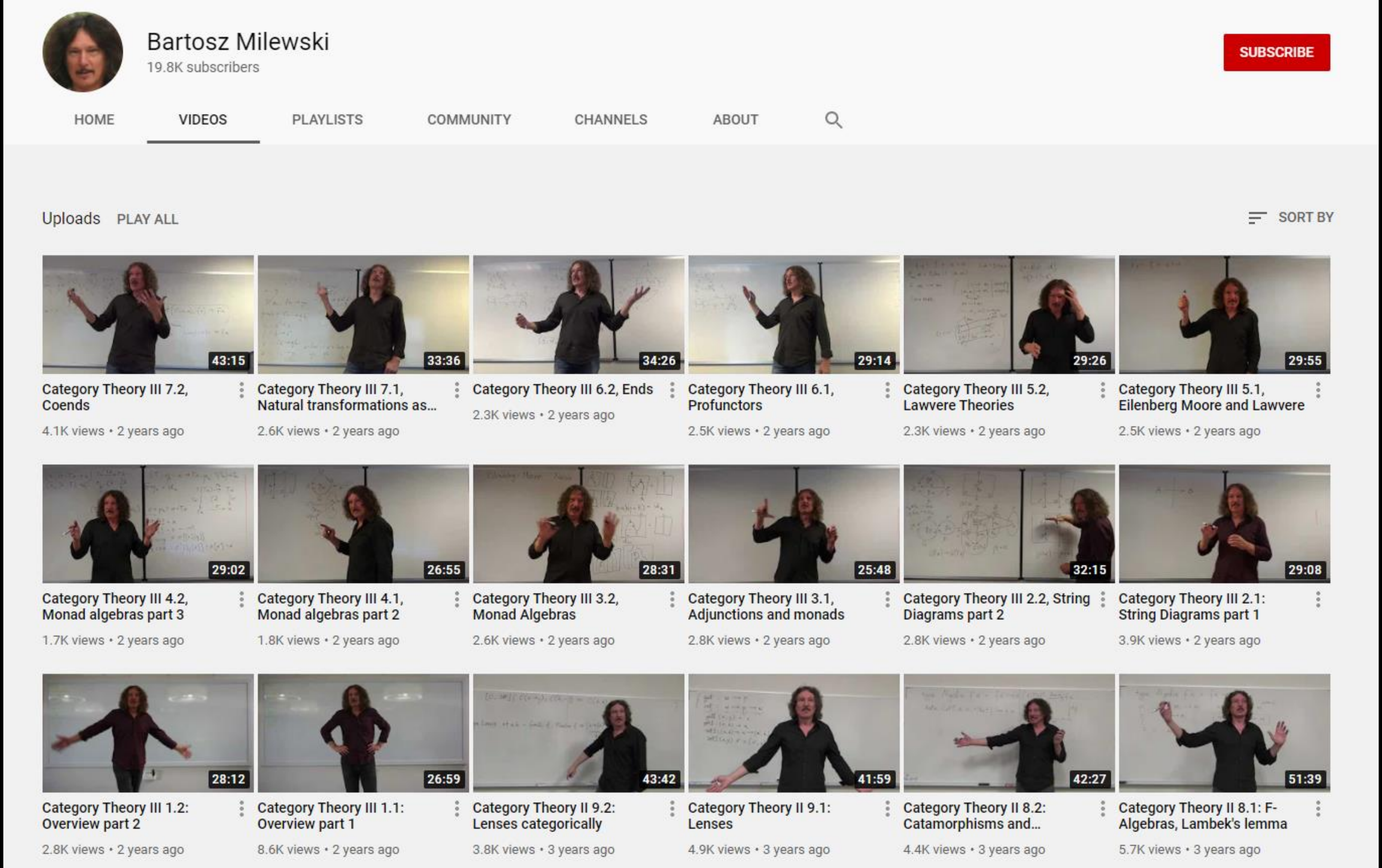
```cpp
auto memoize(auto fn) {
    return [done = std::map<int,int>{}, fn](auto n) mutable  {
        if (auto it = done.find(n); it != done.end())
            return it->second;
        auto const val = fn(n);
        done[n] = val;
        return val;
    };
}
```

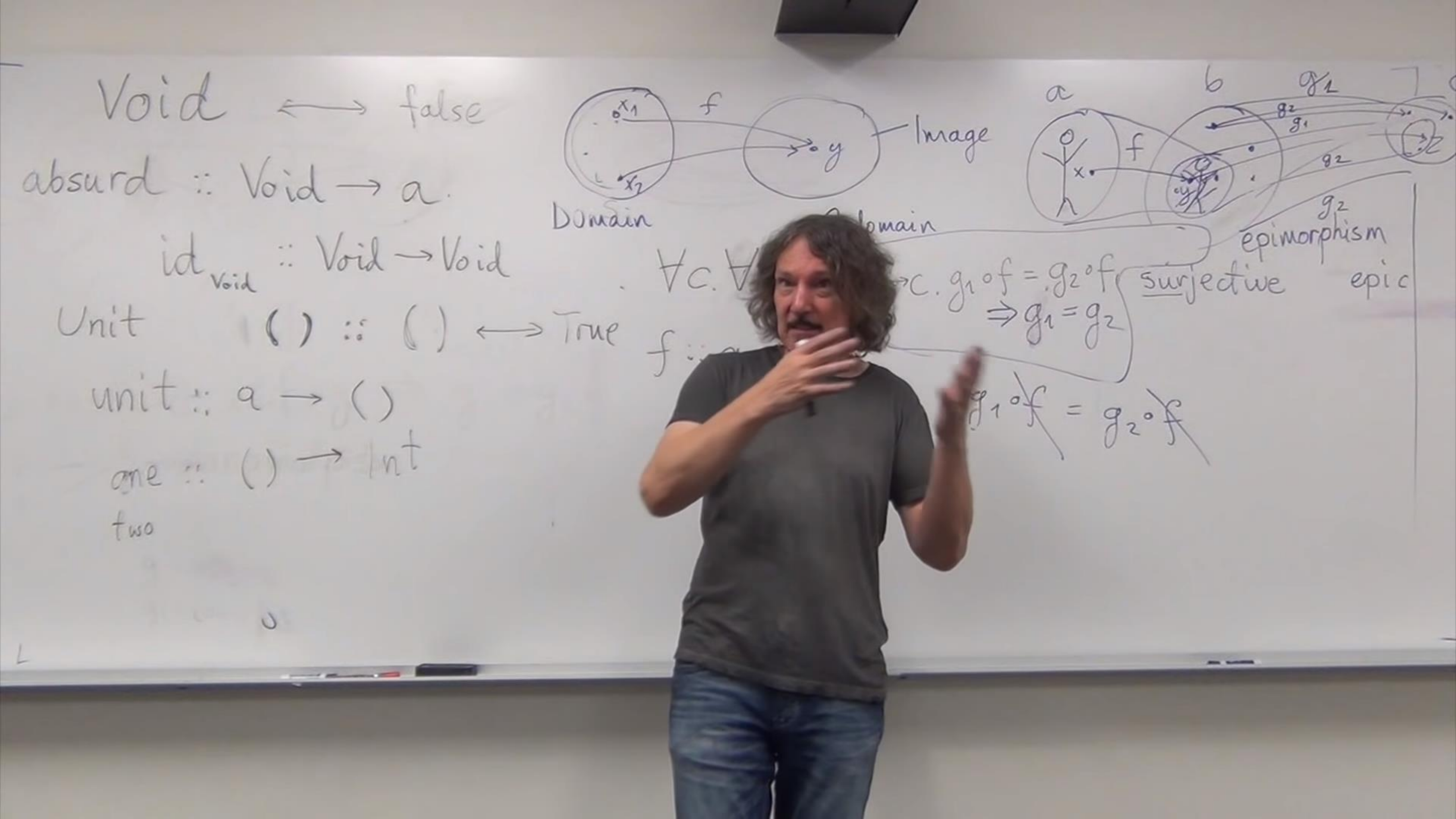
```cpp
auto memoize(auto fn) {
    return [done = std::map<int,int>{}, fn](auto n) mutable  {
        if (auto it = done.find(n); it != done.end())
            return it->second;
        return done[n] = fn(n);
    };
}
```

5. How many different functions are there from `Bool` to `Bool`? Can you implement them all?

```cpp
auto same_(bool b) -> bool { return b;      }
auto diff_(bool b) -> bool { return !b;     }
auto true_(bool b) -> bool { return true;   }
auto fals_(bool b) -> bool { return false; }
```

```
same  ← ⊢
diff  ← ~
true  ← 1∘⊣
false ← 0∘⊣
```


Bartosz Milewski
19.8K subscribers
SUBSCRIBE
HOME
VIDEOS
PLAYLISTS
COMMUNITY
CHANNELS
ABOUT
Uploads
PLAY ALL
SORT BY
43:15
Category Theory III 7.2, Coends
4.1K views • 2 years ago
33:36
Category Theory III 7.1, Natural transformations as...
2.6K views • 2 years ago
34:26
Category Theory III 6.2, Ends
2.3K views • 2 years ago
29:14
Category Theory III 6.1, Profunctors
2.5K views • 2 years ago
29:26
Category Theory III 5.2, Lawvere Theories
2.3K views • 2 years ago
29:55
Category Theory III 5.1, Eilenberg Moore and Lawvere
2.5K views • 2 years ago
29:02
Category Theory III 4.2, Monad algebras part 3
1.7K views • 2 years ago
26:55
Category Theory III 4.1, Monad algebras part 2
1.8K views • 2 years ago
28:31
Category Theory III 3.2, Monad Algebras
2.6K views • 2 years ago
25:48
Category Theory III 3.1, Adjunctions and monads
2.8K views • 2 years ago
32:15
Category Theory III 2.2, String Diagrams part 2
2.8K views • 2 years ago
29:08
Category Theory III 2.1: String Diagrams part 1
3.9K views • 2 years ago
28:12
Category Theory III 1.2: Overview part 2
2.8K views • 2 years ago
26:59
Category Theory III 1.1: Overview part 1
8.6K views • 2 years ago
43:42
Category Theory II 9.2: Lenses categorically
3.8K views • 3 years ago
41:59
Category Theory II 9.1: Lenses
4.9K views • 3 years ago
42:27
Category Theory II 8.2: Catamorphisms and...
4.4K views • 3 years ago
51:39
Category Theory II 8.1: F-Algebras, Lambek's lemma
5.7K views • 3 years ago



Void ⟷ false
absurd :: Void → a
id Void :: Void → Void
Unit
( ) :: ( ) ⟷ True
unit :: a → ( )
one :: ( ) → Int
two
f
Image
Domain
Codomain
a
b
f
g1
g2
epimorphism
surjective
epic
g1 ∘ f = g2 ∘ f
⇒ g1 = g2

C++

Java

Meetup