How to Write PHP like a Haskell Programmer

By Richard


This language is nice
I wonder w
HASKELL!
HASKELL IS AWESOME
EVERYONE SHOULD USE HASKELL
FALSEKNEES.COM © 2016

“A language that doesn't affect the way you think about programming, is not worth knowing.”

- Alan Perlis

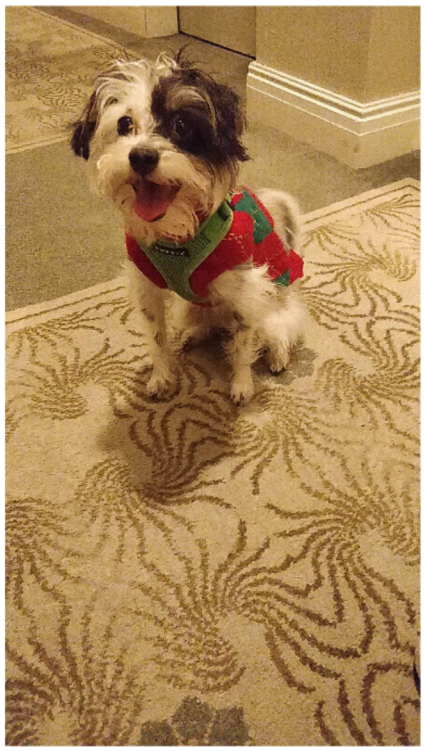
Three Haskelly Catchphrases

1. *"Make business logic errors into type errors"*
2. *"Make invalid states unrepresentable"*
3. *"Constraints liberate, and liberties constrain"*

Three Haskell Features/Techniques

1. Newtypes
2. Algebraic Datatypes
3. Parametric Polymorphism (AKA generics, AKA templates)

Three Dog Pictures



“Make business logic errors into type errors”

~ Haskeller Matt Parsons

Matt Parsons 🏳️‍🌈
@mattoflambda

Following

when people say "but most business logic bugs aren't type errors" i just want to show them how to make bugs into type errors

10:36 AM - 18 Jun 2018

49 Retweets **148** Likes

11 49 148


String -> String -> String -> String
Types are a lie

Newtypes

- Newtypes create a new type by “wrapping” another type.
- They let you restrict or redefine the operations that can legally be performed on that type.
- Classic example: units of measure

```haskell
data Video = Video {
  duration :: Int
}
```

```
*VideoDuration> prettyDuration 200
"3:20"
*VideoDuration> prettyDuration 3500
"58:20"
*VideoDuration> prettyDuration 700000
"194:26:40"
```

```haskell
prettyDuration :: Int -> String
prettyDuration duration = undefined
```

```haskell
prettyDuration :: Int -> String
prettyDuration duration = prettyHours <> prettyMinutes <> prettySeconds
  where
    seconds = duration `mod` 60
    minutes = (duration `div` 60) `mod` 60
    hours = (duration `div` 60 `div` 60)
    prettySeconds
      | hours > 0 || minutes > 0 = pad seconds
      | otherwise = show seconds
    prettyMinutes
      | hours > 0 = pad minutes <> ":"
      | minutes > 0 = show minutes <> ":"
      | otherwise = ""
    prettyHours
      | hours > 0 = (show hours) <> ":"
      | otherwise = ""
    pad n
      | n < 1 = "00"
      | n < 10 = "0" <> (show n)
      | otherwise = show n
```

```
data Video = Video {
  duration :: Int
}
```

```
*VideoDuration> prettyDuration 200
"3:20"
*VideoDuration> prettyDuration 3500
"58:20"
*VideoDuration> prettyDuration 700000
"194:26:40"
```

```
prettyDuration :: Int -> String
prettyDuration duration = undefined
```

```
prettyDurationForVideo :: Video -> String
prettyDurationForVideo (Video duration)
  = prettyDuration duration
```

```haskell
data Video = Video {
  duration :: Int
}
```

But what if this is in milliseconds?

This is seconds

```haskell
prettyDuration :: Int -> String
prettyDuration duration = undefined
```

```haskell
prettyDurationForVideo :: Video -> String
prettyDurationForVideo (Video duration)
  = prettyDuration duration
```

So this is a bug.

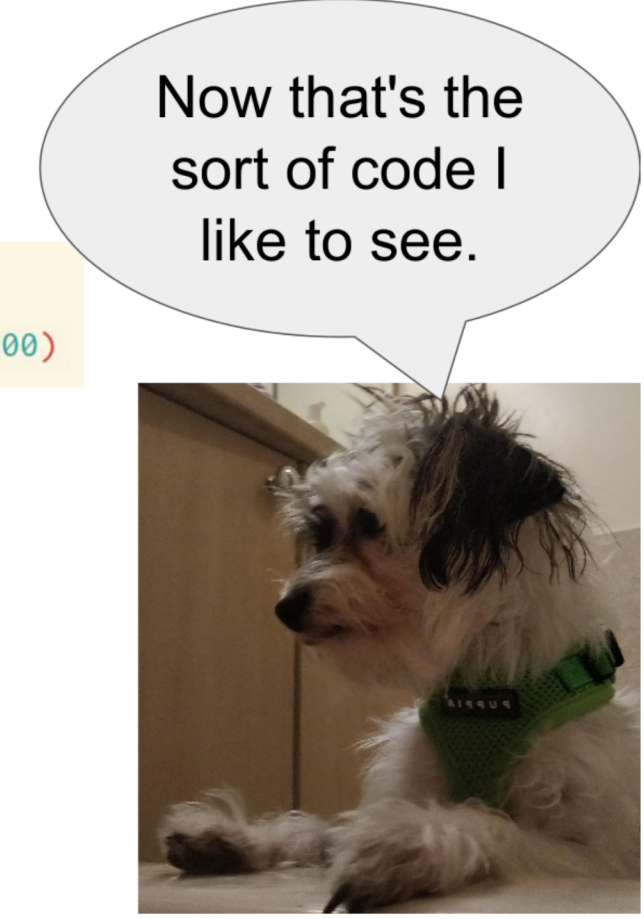
```haskell
toSeconds :: Int -> Int
toSeconds ms = ms `div` 1000
```

```haskell
prettyDurationForVideo :: Video -> String
prettyDurationForVideo (Video duration)
  = prettyDuration (toSeconds duration)
```

```haskell
newtype Seconds = Seconds Int
newtype Milliseconds = Milliseconds Int
```

```haskell
toSeconds :: Milliseconds -> Seconds
toSeconds (Milliseconds ms) = Seconds (ms `div` 1000)
```

```haskell
data Video = Video {
  duration :: Milliseconds
}

prettyDuration :: Seconds -> String
prettyDuration (Seconds duration) = undefined


prettyDurationForVideo :: Video -> String
prettyDurationForVideo (Video duration)
  = prettyDuration (toSeconds duration)
```



So what about PHP?

```php
class Video {
    /** @var int */
    public $duration;

    /** @param int $duration */
    function __construct($duration) {
        $this->duration = $duration;
    }
}

/**
 * @param int $duration
 * @return string
 */
function prettyDuration ($duration) {
    return "";
}
```

- Newtype is just a type with a value inside it.
- In PHP, write a class with a value inside it.

```php
// newtype Milliseconds = Milliseconds Int
class Milliseconds {
    /** @var int */
    public $value;
    /** @param int $value */
    function __construct($value) {
        $this->value = $value;
    }
}
```

```php
// newtype Milliseconds = Milliseconds Int
class Milliseconds {
    /** @var int */
    public $value;
    /** @param int $value */
    function __construct($value) {
        $this->value = $value;
    }
}

// newtype Seconds = Seconds Int
class Seconds {
    /** @var int */
    public $value;
    /** @param int $value */
    function __construct($value) {
        $this->value = $value;
    }
}
```

```php
/**
 * @param Seconds $duration
 * @return PrettyDuration
 */
function prettyDuration ($duration) {
    return new PrettyDuration(
        "" /* insert real implementation here */
    );
}
```

```php
/**
 * @param Milliseconds $ms
 * @return Seconds
 */
function toSeconds ($ms) {
    return new Seconds($ms->value * 1000);
}

class Video {
    /** @var Milliseconds */
    public $duration;

    /** @param Milliseconds $duration */
    function __construct($duration) {
        $this->duration = $duration;
    }
}
```

- Yay! We made a business logic error into a type error!
- So what?
  - Faster feedback
  - Greater maintainability
  - Clearer intent

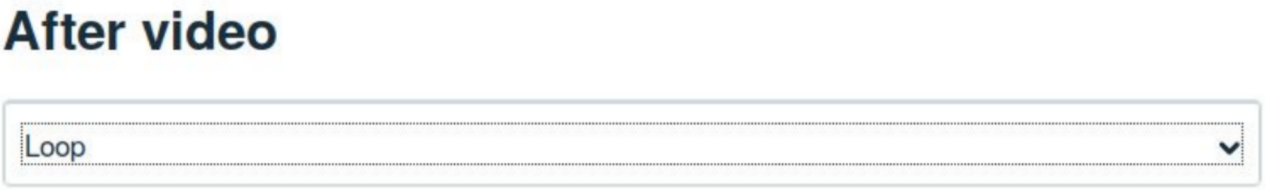
Other applications?

- Units of measure
- Timestamp formats
  - Vimeo Live had a terrible “time-travelling event bug” caused by saving a string representing a timestamp to MySQL, but the format was incompatible with how MySQL interprets time zones.
  - newtype MysqlTimestamp = MysqlTimestamp String
- XSS sanitization?
  - newtype Escaped = Escaped String
  - (Although, this would be basically impossible to enforce in the .phtml paradigm, since rendering .phtml is not done by passing input to a function with a type signature)

“Make invalid states unrepresentable”

~ OCamler Yaron Minsky

Example: Video Outros



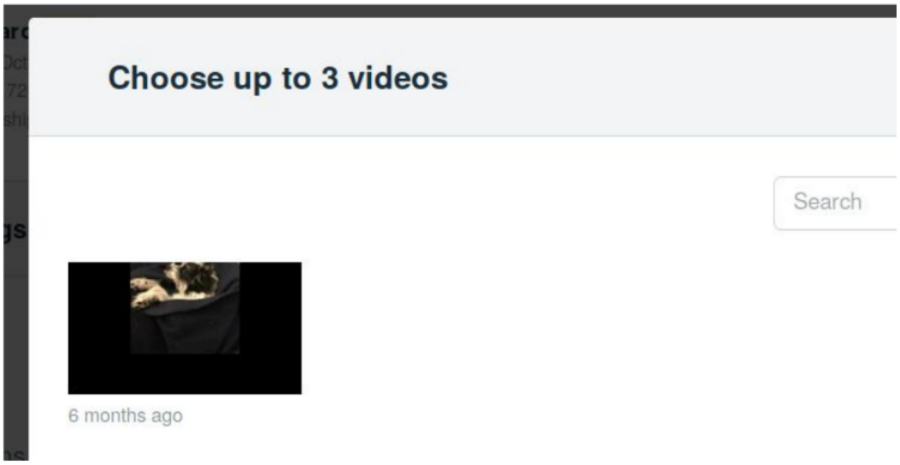
Example: Video Outros

After video

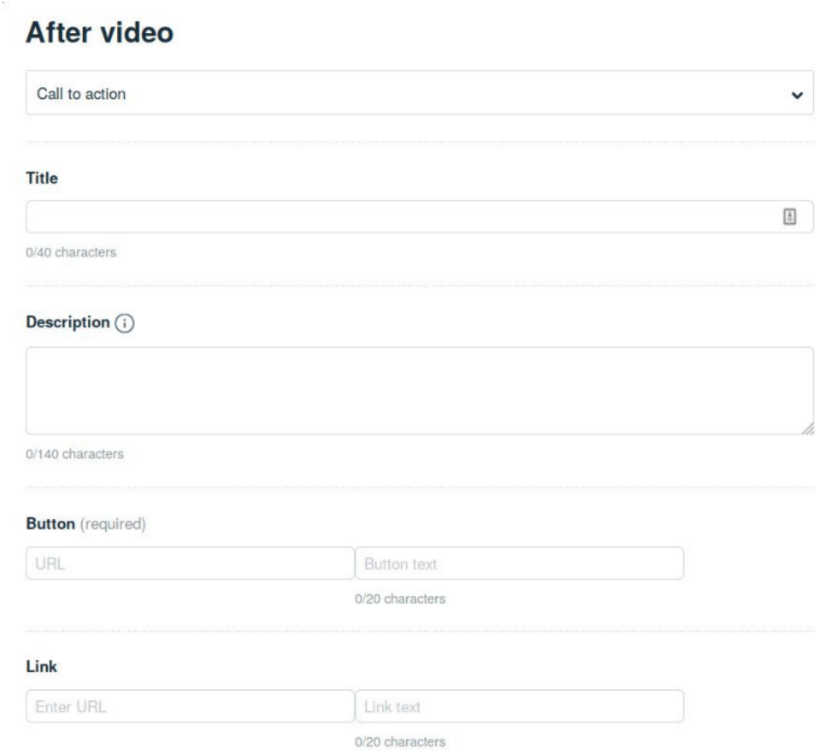
More videos

Shows your three most recent public videos

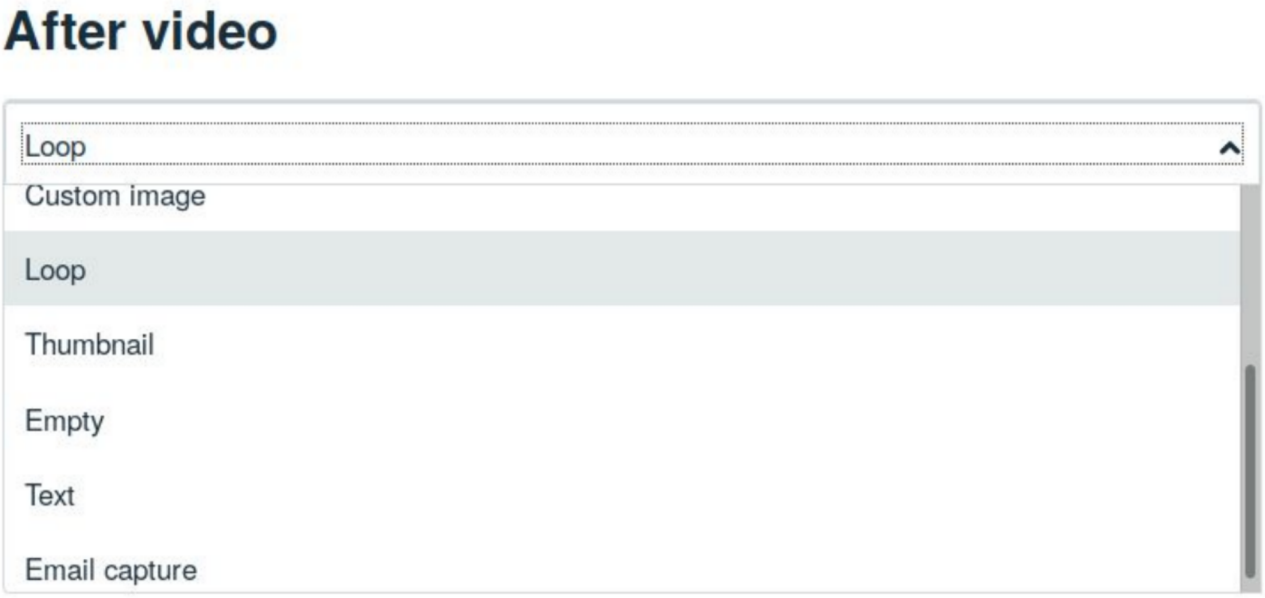
Select specific videos



Example: Video Outros



Example: Video Outros

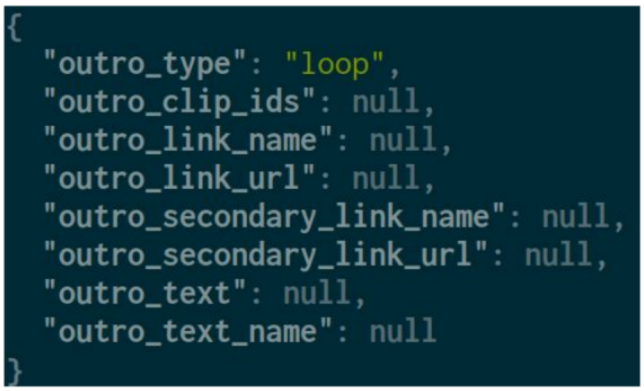


Example: Video Outros

```php
class VideoOutro {
    /** @var string */
    protected $outro_type;

    const OUTRO_TYPE_LOOP = 'loop';

    const OUTRO_TYPE_THREEVIDEOS = 'videos';
    /** @var ?array<int> */
    protected $outro_clip_ids;

    const OUTRO_TYPE_LINK = 'link'; // "Call to Action"
    /** @var ?string */
    protected $outro_link_name;
    /** @var ?string */
    protected $outro_link_url;
    /** @var ?string */
    protected $outro_secondary_link_name;
    /** @var ?string */
    protected $outro_secondary_link_url;
    /** @var ?string */
    protected $outro_text;
    /** @var ?string */
    protected $outro_text_name;
}
```

- All of these properties are nullable.
- But null is not always ok.
- Depending on the value of “outro_type”, null could in fact be forbidden, or mandatory.

Example: Video Outros



```
{
  "outro_type": "loop",
  "outro_clip_ids": null,
  "outro_link_name": null,
  "outro_link_url": null,
  "outro_secondary_link_name": null,
  "outro_secondary_link_url": null,
  "outro_text": null,
  "outro_text_name": null
}
```

VALID

```
{
  "outro_type": "threevideos",
  "outro_clip_ids": [
    123,
    456
  ],
  "outro_link_name": null,
  "outro_link_url": null,
  "outro_secondary_link_name": null,
  "outro_secondary_link_url": null,
  "outro_text": null,
  "outro_text_name": null
}
```

VALID

Example: Video Outros

```
{
  "outro_type": "loop",
  "outro_clip_ids": [
    123,
    456
  ],
  "outro_link_name": null,
  "outro_link_url": null,
  "outro_secondary_link_name": null,
  "outro_secondary_link_url": null,
  "outro_text": null,
  "outro_text_name": null
}
```

INVALID

```
$ php embedsettings.php | jq '.'
{
  "outro_type": "link",
  "outro_clip_ids": null,
  "outro_link_name": null,
  "outro_link_url": null,
  "outro_secondary_link_name": "foo",
  "outro_secondary_link_url": "bar",
  "outro_text": null,
  "outro_text_name": null
}
```

INVALID

What do we do?

- Check at runtime to make sure users never try to create something bad.

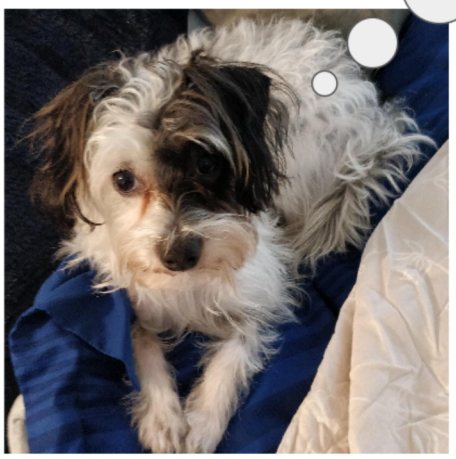
```
if ($this->outro_type === ClipEmbedSettings::OUTRO_TYPE_THREEVIDEOS) {
    $user_clips = Clip::getAllClipsForUser($clip_user, $clip_user);

    $this->requireField('outro_clip_ids');
```

- But what if we could completely eliminate the possibility of invalid outros by construction?

“Algebraic Data Types”

Ways of combining simple types into more complicated types

- Sum types
- Product types



What are those? It sounds like math...

You already know Product Types

- Most languages let you define your own “product types”.
- Examples: structs, classes, tuples
- A product type has something AND something else, at the same time.

```php
class LinkOutro {
    /** @var string */
    public $name;
    /** @var string */
    public $url;
    /** @var ?string */
    public $secondary_name;
    /** @var ?string */
    public $secondary_url;
    /** @var ?string */
    public $text;
    /** @var ?string */
    public $text_name;
}
```

```go
type LinkOutro struct {
    Name string
    Url string
    SecondaryName *string
    SecondaryURL *string
    Text *string
    TextName *string
}
```

You already know Sum Types

- All languages have Sum Types
- Only cool languages let you define your own.
- A sum type is something OR it is something else. Not at the same time. Like Enums.

```
data Boolean
  = True
  | False
```

```
data Integer
  = 1
  | 2
  | 3
  | ...
  | MAX_INT
  | -1
  | -2
  | ...
  | MIN_INT
```

```
data OutroType
  = Loop
  | Link
  | ThreeVideos
```

Way cooler than Enums, because you can have Sums of Products.

```
data Shape
  = Rectangle { height: Float, width: Float }
  | Circle { radius: Float }

area :: Shape -> Float
area (Rectangle h w) = h * w
area (Circle r) = pi * r * r
```

VideoOutro as a Sum of Products

```
data Outro
  = Loop
  | Link {
      name :: String,
      url :: String,
      secondary_name :: Maybe String,
      secondary_url :: Maybe String,
      text :: Maybe String,
      text_name :: Maybe String
    }
  | ThreeVideos { video_ids :: [Int] }
```

Sum types can be expressed using OOP inheritance

```php
abstract class VideoOutro {}

class LoopOutro extends VideoOutro {}

class ThreeVideosOutro extends VideoOutro {
    /** @var array<int> */
    public $video_ids;
}

class LinkOutro extends VideoOutro {
    /** @var string */
    public $name;
    /** @var string */
    public $url;
    /** @var ?string */
    public $secondary_name;
    /** @var ?string */
    public $secondary_url;
    /** @var ?string */
    public $text;
    /** @var ?string */
    public $text_name;
}
```

Exhaustiveness Checking for Sum Types

Let's serialize our outro to a JSON object, for the browser javascript to read.

```
instance JSON Outro where
  showJSON :: Outro -> JSValue
  showJSON outro = case outro of
    Loop -> makeObj [("type", showJSON "loop")]
    ThreeVideos v_ids -> makeObj
      [ ("type", showJSON "three_videos")
      , ("video_ids", showJSON v_ids )
      ]
```

- Oops! We forgot about "Link".
- Haskell compiler will helpfully point this out!

```
Main.hs:49:20: error: [-Wincomplete-patterns, -Werror=incomplete-patt
    Pattern match(es) are non-exhaustive
    In a case alternative: Patterns not matched: (Link _ _ _ _ _ _)
   |
49 |   showJSON outro = case outro of
   |                    ^^^^^^^^^^^^...
```

Exhaustiveness Checking for Subclasses?

- Not a thing.

```php
/**
 * @param VideoOutro $outro
 * @return string
 */
function serializeToJson ($outro) {
    switch (get_class($outro)) {
    case LoopOutro::class:
        return json_encode([
            "type" => "loop"
        ]);
    case ThreeVideosOutro::class:
        return json_encode([
            "type" => "three_videos",
            "video_ids" => $outro->video_ids
        ]);
    }
    return "";
}
```


- Psalm forces you to have a fallback, because it is not clever enough (yet?) to see if you handled all possible subclasses. Stay tuned!

Typescript has union types that support exhaustiveness checking

```
interface LoopOutro { "outro_type": "loop" }
interface ThreeVideosOutro { "outro_type": "three_videos", "video_ids": Array<Number> }
interface LinkOutro {
    outro_type : "link",
    name: string,
    url: string,
    secondary_name?: string,
    secondary_url?: string,
    text?: string,
    text_name?: string
}
type VideoOutro
    = LoopOutro
    | ThreeVideosOutro
    | LinkOutro
```

Typescript has union types that support exhaustiveness checking

```
17 function checkExhaustive(x: never): never {
18     throw new Error("Exhaustiveness check fails for: " + x)
19 }
20
21 function displayOutro (outro: VideoOutro): string {
22     if (outro.outro_type === "loop") {
23         return "<LoopOutro/>"
24     } else if (outro.outro_type === "three_videos") {
25         return `<ThreeVideosOutro videoIds=${outro.video_ids}`
26     }
>> 27     checkExhaustive(outro)
```

Advantages?

- If you add a new member to your sum type, you will get a bunch of type errors. Once you've fixed them, you can be confident you've handled it everywhere you need to if you fix all the type errors.
- For example, we recently added a new "clip privacy setting", "CLIP_PRIVACY_STOCK_FOOTAGE".
  - There are 41 places in the codebase where "CLIP_PRIVACY_STOCK_FOOTAGE" occurs, in a dozen different files.
  - I didn't implement this -- but I bet it was a very tedious and manual process for the developers who did.
- All properties that in our database are encoded as "enums" should be encoded as sum types in our code.

“Constraints liberate, and liberties constrain”

~ Scalaer Rúnar Bjarnason

```
f :: Bool -> Bool
g :: a -> a
```

Parametric Polymorphism
(AKA generics)

f and g are **pure** functions (they do not depend on anything but the inputs described in their type signature, and do not perform any action but returning an output)

Counting the number of functions from Bool to Bool

Name	Input	Output
const true	True	True
	False	True
const false	True	False
	False	False
id	True	True
	False	False
not	True	False
	False	True

If you know type a, and you know type b, then the number of functions from a to b is B^A

Counting the number of functions from a to a

(where a is a generic type parameter)

Name	Input	Output
id	a	a

- We “liberated” our types, which “constrained” the number of possibilities
- The type is now a more complete description of the function.

```
f :: (Int, Int) -> Bool
g :: (a,   a)   -> Bool
```

- 2^(2^64) = 2^(18446744073709551616) possibilities for f
- 2 possibilities for g
  - \x -> true
  - or \x -> false
- Those are pretty pointless functions.
- We've "liberated" it too much, can we add back a constraint?

```
g :: (Eq a) => (a, a) -> Bool
```

Equality “Typeclass”
like saying “a implements the ‘equals’ interface”

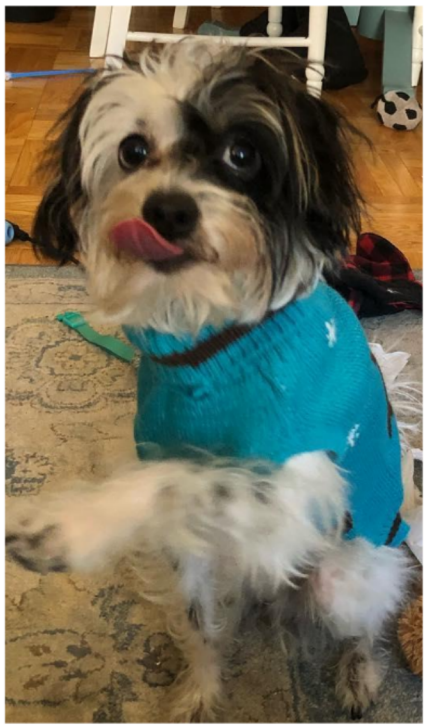
Name	a1 == a2	Output
!=	True	False
	False	True
==	True	True
	False	False
const true	True	True
	False	True
const false	True	False
	False	False

So what?

- Code to general interfaces, not concrete implementations
- We already knew this. OOP theorists told us this helps reduce coupling.
- Still, generic type signatures look complicated and feel like overengineering.
- But they make things simpler, at least in this abstract mathematical sense
- And your types will be a more complete description of your code.

Inertia

- Types, if you take full advantage of them, can be a wonderful tool to keep your code maintainable and robust.
- Most code at Vimeo was written before we had a type system.
- Taking full advantage of our type system will require a deliberate choice.
    - Make business logic errors into type errors
    - Make invalid states unrepresentable
    - Liberties constrain, constraints liberate



Thank you!

Thanks to Matt & Collin & r/haskell for help with this talk