Make It Purple

An Introduction to Type Level Programming

Rebecca Skinner

September 4, 2021

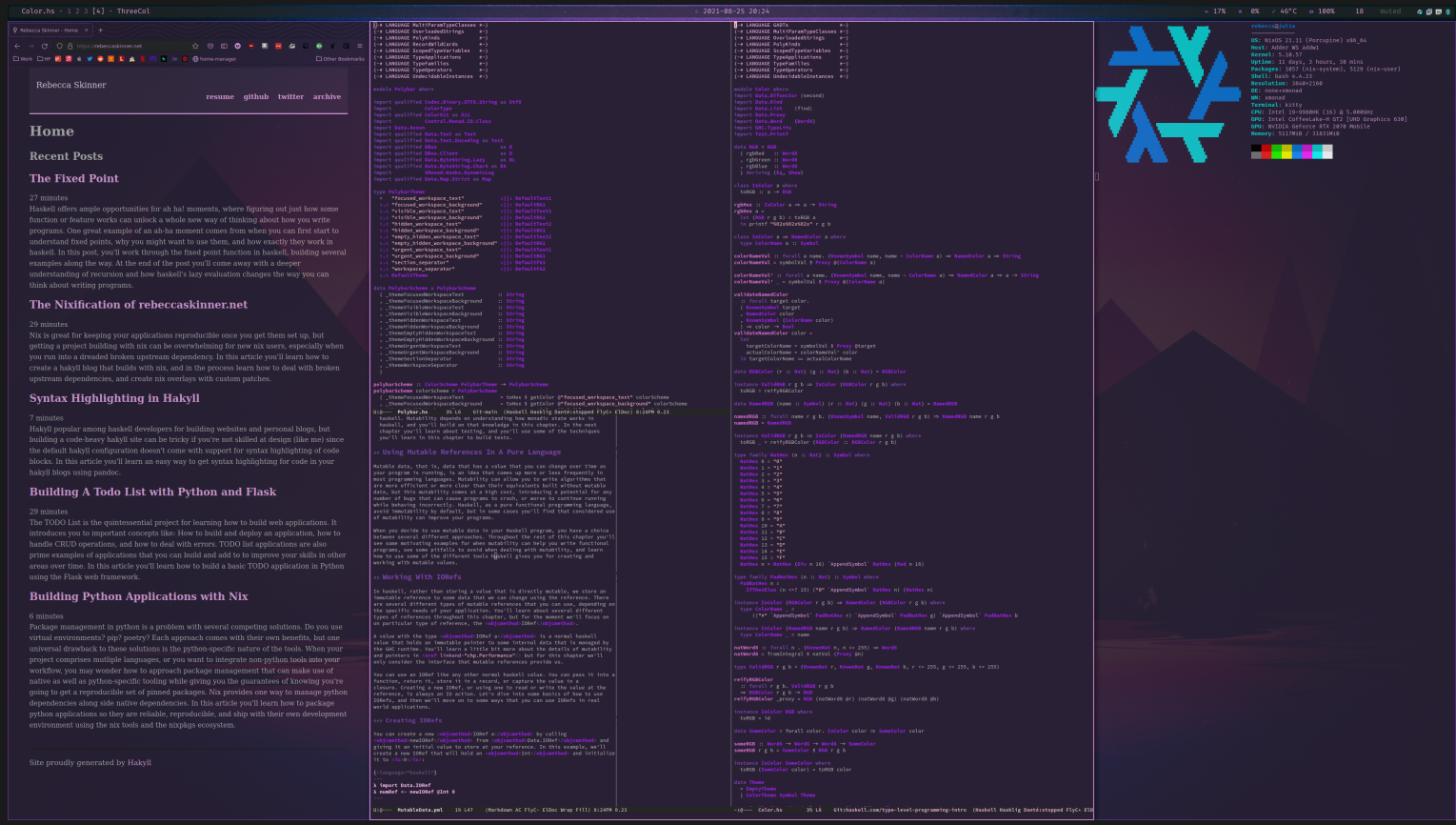
Prelude

Hello, World

- Rebecca Skinner: Me
- Mercury: My Employer (We're Hiring)
- The Ideas and Contents of this Talk: My Own
- Effective Haskell: My Book! (Est. late 2021 / early 2022)
- @cercerilla: Twitter
- Slides and Code: https://rebeccaskinner.net

Building A Theming Engine for XMonad

XMonad: Purple Edition



Let's Build It!

On the surface, a theming system doesn't need to be very complicated.

Let's Build It!

On the surface, a theming system doesn't need to be very complicated.

```
polybarColorScheme = PolybarColorScheme
  { focusedWorkspaceText         = "#dda0dd"
  , focusedWorkspaceBackground   = "#2a2035"
  , visibleWorkspaceText         = "#dda0dd"
  , visibleWorkspaceBackground   = "#2a2035"
  }
```

Let's Build It!

On the surface, a theming system doesn't need to be very complicated.

```
polybarColorScheme = PolybarColorScheme
  { focusedWorkspaceText         = "#dda0dd"
  , focusedWorkspaceBackground   = "#2a2035"
  , visibleWorkspaceText         = "#dda0dd"
  , visibleWorkspaceBackground   = "#2a2035"
  }
```

But now a change to our theme means changing every individual element.

```
stylishConfig :: XConfig a -> XConfig a
stylishConfig cfg = cfg
  { normalBorderColor  = "#dda0dd"
  , focusedBorderColor = "#2a2035"
  }
```

With A Color Palette

One solution to allow us to uniformly change our color scheme would be to use a *color palette*.

With A Color Palette

One solution to allow us to uniformly change our color scheme would be to use a *color palette*.

```haskell
data RGB = RGB
  { rgbRed   :: Word8
  , rgbGreen :: Word8
  , rgbBlue  :: Word8
  } deriving (Eq, Show)

type ColorPalette = Map Text RGB

defaultPalette = fromList $
  [ ("foreground", RGB 0x3a 0x20 0x35)
  , ("background", RGB 0xdd 0xa0 0xdd) ]
```

Theming With A Color Palette

Now we can consistently reference colors by their name.

```
polybarColorScheme :: ColorPalette -> Maybe PolybarColorScheme
polybarColorScheme theme = PolybarColorScheme
  <$> theme !? "foreground"
  <*> theme !? "background"
  <*> theme !? "foreground"
  <*> theme !? "background"
```

Missing Information

This buys us some consistency, but we still have problems:

Missing Information

This buys us some consistency, but we still have problems:

1. Every component needs to know ahead of time what keys are available in the palette
2. We can't easily tell what parts of a palette are being used, making maintenance harder
3. We can't statically guarantee that our theme will work

Missing Information

This buys us some consistency, but we still have problems:

1. Every component needs to know ahead of time what keys are available in the palette
2. We can't easily tell what parts of a palette are being used, making maintenance harder
3. We can't statically guarantee that our theme will work

Runtime errors can be painful when we're configuring a desktop environment, because it might mean that we have to fix everything from a terminal.

Using The Type System To Improve Our Theming System

Needs More Type System

Let's look at how we can use the type system to help make our theming system easier to use.

Needs More Type System

Let's look at how we can use the type system to help make our theming system easier to use. Here's what we'd like to do:

Needs More Type System

Let's look at how we can use the type system to help make our theming system easier to use. Here's what we'd like to do:

- Ensure that the type of any function that uses theme elements tells what elements it uses

Needs More Type System

Let's look at how we can use the type system to help make our theming system easier to use. Here's what we'd like to do:

- Ensure that the type of any function that uses theme elements tells what elements it uses
- Ensure that we can't pass a theme to a function if it's missing required elements

But First, A Demo

Our `colorDemo` function lets us get colors from a `theme` by name. Instead of passing the color we want as a string, we're using *visible type applications* to pass in the color as a *type*.

```
{-# LANGUAGE TypeApplications #-}

colorDemo theme =
  let r = lookupColor @"red" theme
      g = lookupColor @"green" theme
      b = lookupColor @"blue" theme
  in show (r,g,b)
```

But First, A Demo

The type of `colorDemo` can be inferred for us, and tells us exactly which colors must be available in our theme.

```
{-# LANGUAGE TypeApplications #-}

colorDemo
  :: ( HasColor "red" theme
     , HasColor "green" theme
     , HasColor "blue" theme )
  => ThemeInstance theme -> String
colorDemo theme =
  let r = lookupColor @"red" theme
      g = lookupColor @"green" theme
      b = lookupColor @"blue" theme
  in show (r,g,b)
```

Let's Make A Theme Instance

Our demo referenced a `ThemeInstance` type that holds theme information, so let's make one.

Let's Make A Theme Instance

Our demo referenced a `ThemeInstance` type that holds theme information, so let's make one.

```
newtype ThemeInstance                    =
  ThemeInstance { getThemeInstance :: Map String RGB }
  deriving Show
```

A Theme Instance With Phantom Types

We need to keep track of all of the theme elements that belond to the theme somehow.

A Theme Instance With Phantom Types

We need to keep track of all of the theme elements that belond to the theme somehow.

We can use a *phantom type* to hold the list of colors in our theme.

A Theme Instance With Phantom Types

We need to keep track of all of the theme elements that belond to the theme somehow.

We can use a *phantom type* to hold the list of colors in our theme.

```
newtype ThemeInstance  theme           =
  ThemeInstance { getThemeInstance :: Map String RGB }
  deriving Show
```

A Phantom Type With a Theme Kind

The theme that we track with our `ThemeInstance` shouldn't be just *any* type though. Most types wouldn't make sense as a theme. What would `ThemeInstance Int` even be?
We can constrain `theme` by giving it a *Kind Signature*. Here we're saying the kind of `theme` must be `Theme`.

A Phantom Type With a Theme Kind

The theme that we track with our `ThemeInstance` shouldn't be just *any* type though. Most types wouldn't make sense as a theme. What would `ThemeInstance Int` even be?
We can constrain `theme` by giving it a *Kind Signature*. Here we're saying the kind of `theme` must be `Theme`.

```
newtype ThemeInstance (theme :: Theme) =
  ThemeInstance { getThemeInstance :: Map String RGB }
  deriving Show
```

How Kind of You

When we're programming at the value level, we tend to think in terms of *types* and *values*. A type has *inhabitants* that are plain haskell values. For example, the `Bool` type has two inhabitants: `True` and `False`.

How Kind of You

When we're programming at the value level, we tend to think in terms of *types* and *values*. A type has *inhabitants* that are plain haskell values. For example, the `Bool` type has two inhabitants: `True` and `False`. A kind is analogous to a type, but where the inhabitants of a type are values, the inhabitants of a kind are types. In other words, *kinds are the types of types*.

Defining A Theme At The Type Level

What Is a Theme?

We said that `ThemeInstance` has a phantom type parameter with the kind `Theme`, but what is a `Theme` anyway?

- A collection of colors

What Is a Theme?

We said that `ThemeInstance` has a phantom type parameter with the kind `Theme`, but what is a `Theme` anyway?

- A collection of colors
- Known at compile time

What Is a Theme?

We said that `ThemeInstance` has a phantom type parameter with the kind `Theme`, but what is a `Theme` anyway?

- A collection of colors
- Known at compile time
- Identifiable with a name like *"red"* or *"foreground"*

Colors At The Type Level

A `Theme` is a collection of type-level colors, so we need to define what a color looks like at the type level.

Theme Elements By Name

We want to refer to theme elements by name (*"red"*, *"green"*, *"blue"*).
We could define types for all the colors:

```
data Red
data Green
data Blue
```

Theme Elements By Name

We want to refer to theme elements by name (*"red"*, *"green"*, *"blue"*). We could define types for all the colors:

```
data Red
data Green
data Blue
```

But what a pain! We could try to enumerate every named color, but it would be a nightmare. Instead, let's use *type-level strings*

Symbolism

Symbols are the type-level equivalents to strings. We can type them just like strings, and use them like types.

```
> :type "green"
"green" :: [Char]
> :kind "green"
"green" :: Symbol
```

Symbolism

Symbols are the type-level equivalents to strings. We can type them just like strings, and use them like types.

```
> :type "green"
"green" :: [Char]
> :kind "green"
"green" :: Symbol
```

Symbol literals are all instances the `KnownSymbol` typeclass. This let's us get the value of a `Symbol` at runtime as a string using the `symbolVal` function:

```
> symbolVal $ Proxy @"green"
"green"

> :type symbolVal $ Proxy @"green"
symbolVal $ Proxy @"green" :: String
```

A Theme Of Many Colors

A theme isn't just a single color, it's a collection of colors. A theme type needs to capture all of the colors that belond to the theme. We can do this by creating a *type-level list*. The syntax is the same as it is for creating a regular list:

A Theme Of Many Colors

A theme isn't just a single color, it's a collection of colors. A theme type needs to capture all of the colors that belond to the theme. We can do this by creating a *type-level list*. The syntax is the same as it is for creating a regular list:

```
type Theme = [Symbol]
```

Checking The Colors In A Theme

Now that we know what a `Theme` is, we need to see if a given color is part of a theme. Let's start by looking at a type signature for `lookupColor`:

Checking The Colors In A Theme

Now that we know what a `Theme` is, we need to see if a given color is part of a theme. Let's start by looking at a type signature for `lookupColor`:

```
lookupColor
  :: forall colorName theme.
  ( KnownSymbol colorName
  , HasColor colorName theme)
  => ThemeInstance theme
  -> RGB
```

Implementing `HasColor`

`HasColor` is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

Implementing HasColor

HasColor is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

```
class HasColor (color :: Symbol) (container :: Theme)
```

Implementing HasColor

`HasColor` is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

```
class HasColor (color :: Symbol) (container :: Theme)
```

We're going to define two instances of `HasColor`. We'll start with an instance that will serve as the *base case* for our type level program.

Implementing HasColor

`HasColor` is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

```
class HasColor (color :: Symbol) (container :: Theme)
```

We're going to define two instances of `HasColor`. We'll start with an instance that will serve as the *base case* for our type level program.

```
instance HasColor color (color : colors)
```

Implementing HasColor

`HasColor` is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

```
class HasColor (color :: Symbol) (container :: Theme)
```

We're going to define two instances of `HasColor`. We'll start with an instance that will serve as the *base case* for our type level program.

```
instance HasColor color (color : colors)
```

Next we'll define a recursive case. If the head of our theme list doesn't match the target color, then there is still a valid instance if the rest of the theme matches.

Implementing HasColor

`HasColor` is a multi-parameter typeclass with two arguments. The first is the color we want to validate is in the theme, and the second is the theme we want to check.

```
class HasColor (color :: Symbol) (container :: Theme)
```

We're going to define two instances of `HasColor`. We'll start with an instance that will serve as the *base case* for our type level program.

```
instance HasColor color (color : colors)
```

Next we'll define a recursive case. If the head of our theme list doesn't match the target color, then there is still a valid instance if the rest of the theme matches.

```
instance                          (HasColor color colors)
  => HasColor color (currentColor : colors)
```

Dealing with Overlapping Instances

If we try to use `HasColor` right now, we'll run into a problem: our two instances seem to overlap if `color` is the same as `currentColor`.
We can get past the error here by explicitly telling GHC that when there is an overlap, to prefer our other instance:

Dealing with Overlapping Instances

If we try to use `HasColor` right now, we'll run into a problem: our two instances seem to overlap if `color` is the same as `currentColor`.
We can get past the error here by explicitly telling GHC that when there is an overlap, to prefer our other instance:

```
instance {-# OVERLAPPABLE #-} (HasColor color colors)
  => HasColor color (currentColor : colors)
```

Getting A Theme Element

Now that we can ensure a color is part of a theme, let's get one out of a theme instance:

Getting A Theme Element

Now that we can ensure a color is part of a theme, let's get one out of a theme instance:

```
lookupColor
  :: forall colorName theme.
  ( KnownSymbol colorName
  , HasColor colorName theme)
  => ThemeInstance theme -> RGB
lookupColor (ThemeInstance colors) =
  let
    targetName = symbolVal $ Proxy @colorName
  in colors Map.! targetName
```

Demo

```haskell
demoThemeInstance :: ThemeInstance ["red","green","blue"]
demoThemeInstance = ThemeInstance . Map.fromList $
  [("red", RGB 0xff 0x00 0x00)
  ,("green", RGB 0x00 0xff 0x00)
  ,("blue", RGB 0x00 0x00 0xff)]
```

Demo

```
demoThemeInstance :: ThemeInstance ["red","green","blue"]
demoThemeInstance = ThemeInstance . Map.fromList $
  [("red", RGB 0xff 0x00 0x00)
  ,("green", RGB 0x00 0xff 0x00)
  ,("blue", RGB 0x00 0x00 0xff)]
```

If we look up a color that exists in the theme, we get back the right value.

```
> lookupColor @"red" demoThemeInstance
RGB {rgbRed = 255, rgbGreen = 0, rgbBlue = 0}
```

Demo

```
demoThemeInstance :: ThemeInstance ["red","green","blue"]
demoThemeInstance = ThemeInstance . Map.fromList $
  [("red", RGB 0xff 0x00 0x00)
  ,("green", RGB 0x00 0xff 0x00)
  ,("blue", RGB 0x00 0x00 0xff)]
```

If we look up a color that exists in the theme, we get back the right value.

```
> lookupColor @"red" demoThemeInstance
RGB {rgbRed = 255, rgbGreen = 0, rgbBlue = 0}
```

If we try to look up a color not in the theme, we get a compile-time error

```
> lookupColor @"yellow" demoThemeInstance

<interactive>:80:1: error:
    • No instance for (HasColor "yellow" '[])
        arising from a use of ‘lookupColor’
    • In the expression: lookupColor @"yellow" demoThemeInstance
      In an equation for ‘it’:
          it = lookupColor @"yellow" demoThemeInstanc
```

Building A Better Theme Instance

Mind The Gap

Using the type system to track the colors in our theme lets us feel safe, but we have a runtime error lurking:

Mind The Gap

Using the type system to track the colors in our theme lets us feel safe, but we have a runtime error lurking:

```
> lookupColor @"blue" demoThemeInstance
  *** Exception: Map.!: given key is not an element in the map
  CallStack (from HasCallStack):
    error, called at
      libraries/containers/containers/src/Data/Map/Internal.hs:627:17
      in containers-0.6.2.1:Data.Map.Internal
```

An Instance of Weakness

What happened?

An Instance of Weakness

What happened?

```
demoThemeInstance :: ThemeInstance ["red","green","blue"]
demoThemeInstance = ThemeInstance . Map.fromList $
  [("red", RGB 0xff 0x00 0x00)]
```

An Instance of Weakness

What happened?

```
demoThemeInstance :: ThemeInstance ["red","green","blue"]
demoThemeInstance = ThemeInstance . Map.fromList $
  [("red", RGB 0xff 0x00 0x00)]
```

There's nothing ensuring the contents of the map actually match the colors in the `Theme`.

Fixing The Problem

If we want to fix the problem, we need to bring more of the theming system into the type level. We'll start by defining colors at the type level (again).

Colors At The Type Level: Part 2

Earlier we defined colors at the type level as `Symbol` types. Now let's try to capture more information about a color at the type level. Our goal:

Capture all of the relevant information about a color at compile time.

X11 Colors At The Type Level

One way to bring colors to the type level is to define new types for a common set of colors we might want to use. This gives us an easy way to refer to a fixed collection of colors that each have their own type.

```
data RebeccaPurple = RebeccaPurple
```

Type Level RGB

Working with a fixed color pallet can be a nice convenience, but as we noted earlier it can end up being inflexible. We'd like to also provide a way to work with arbitrary RGB colors:

Type Level RGB

Working with a fixed color pallet can be a nice convenience, but as we noted earlier it can end up being inflexible. We'd like to also provide a way to work with arbitrary RGB colors:

```
data RGBColor (r :: Nat) (g :: Nat) (b :: Nat) = RGBColor
```

Color Level At Runtime

If we define color information at the type level, we need a way to get access to it at runtime.

Type Level RGB Is A Color

An `IsColor` typeclass can help us identify things that have a runtime representation as an RGB color:

Type Level RGB Is A Color

An `IsColor` typeclass can help us identify things that have a runtime representation as an RGB color:

```
class IsColor a where
  toRGB :: a -> RGB
```

Type Level RGB Is A Color

An `IsColor` typeclass can help us identify things that have a runtime representation as an RGB color:

```
class IsColor a where
  toRGB :: a -> RGB
```

Creating an instance for our runtime `RGB` records is trivial

Type Level RGB Is A Color

An `IsColor` typeclass can help us identify things that have a runtime representation as an RGB color:

```
class IsColor a where
  toRGB :: a -> RGB
```

Creating an instance for our runtime `RGB` records is trivial

```
instance IsColor RGB where
  toRGB = id
```

Type Level RGB Is A Color

An `IsColor` typeclass can help us identify things that have a runtime representation as an RGB color:

```
class IsColor a where
  toRGB :: a -> RGB
```

Creating an instance for our runtime `RGB` records is trivial

```
instance IsColor RGB where
  toRGB = id
```

As is creating an instance for a color from a pre-defined color pallet.

```
instance IsColor RebeccaPurple where
  toRGB = const $ RGB 0x66 0x33 0x99
```

Making an `RGBColor` Type Into a Runtime Value

Not all types `RGBColor r g b` can map to an `RGB` color. To be able to make a valid color we have to put some constraints on `r`, `g`, and `b`:

- They must be numbers

Making an `RGBColor` Type Into a Runtime Value

Not all types `RGBColor r g b` can map to an `RGB` color. To be able to make a valid color we have to put some constraints on `r`, `g`, and `b`:

- They must be numbers
- With a value between 0 and 255 (inclusive)

Making an `RGBColor` Type Into a Runtime Value

Not all types `RGBColor r g b` can map to an `RGB` color. To be able to make a valid color we have to put some constraints on `r`, `g`, and `b`:

- They must be numbers
- With a value between 0 and 255 (inclusive)

Making an `RGBColor` Type Into a Runtime Value

Not all types `RGBColor r g b` can map to an `RGB` color. To be able to make a valid color we have to put some constraints on `r`, `g`, and `b`:

- They must be numbers
- With a value between 0 and 255 (inclusive)

We can use `ConstraintKinds` to create an alias for this set of constraints:

Making an `RGBColor` Type Into a Runtime Value

Not all types `RGBColor r g b` can map to an `RGB` color. To be able to make a valid color we have to put some constraints on `r`, `g`, and `b`:

- They must be numbers
- With a value between 0 and 255 (inclusive)

We can use `ConstraintKinds` to create an alias for this set of constraints:

```
type ValidRGB r g b =
  ( KnownNat r, KnownNat g, KnownNat b
  , r <= 255, g <= 255, b <= 255)
```

Converting an RGBColor Into A Runtime RGBValue

Now that we can guarantee the parameters to `RGBColor` are valid, we can easily generate a runtime `RGB` value from one.

```
instance ValidRGB r g b => IsColor (RGBColor r g b) where
  toRGB _ = RGB (natWord8 @r) (natWord8 @g) (natWord8 @b)
```

The RGBColor Problem

Now we can create RGBColor types at compile time, but that brings back to mind an old problem:

```
RGBColor @255 @0 @255
```

```
RGB 0xff 0x00 0xff
```

It looks like we're back to the problem of using raw RGB values everywhere.

The Hardest Problem in Computer Science

Having the option of using a pallet of named colors showed some promise. How can we bring that to the type level?

- Naming should be general. It should be able to support our type-level RGB colors, but other types of colors as well.

The Hardest Problem in Computer Science

Having the option of using a pallet of named colors showed some promise. How can we bring that to the type level?

- Naming should be general. It should be able to support our type-level RGB colors, but other types of colors as well.
- Names need be ~Symbol~s, so that we can use them at the type level.

Type Families

When we want to define a function for several different *types*, we often look toward functions defined by typeclasses.
In our case though, we don't want to have a function from one value to another. Instead, we want something to go from one type to another type. For that, we use *Type Families*.

Creating A Type Family For Color Names

Let's create a type family for things that are colors and can be named:

```
class IsColor a => NamedColor a where
  type ColorName a :: Symbol
```

`ColorName` is an *associated type family* that let's us map a type of to another type (of kind `Symbol`).

Creating A Type Family For Color Names

Let's create a type family for things that are colors and can be named:

```
class IsColor a => NamedColor a where
  type ColorName a :: Symbol
```

`ColorName` is an *associated type family* that let's us map a type of to another type (of kind `Symbol`).

Let's use `NamedColor` to write a function that will let us get the name of a given color as a runtime `String`:

Creating A Type Family For Color Names

Let's create a type family for things that are colors and can be named:

```
class IsColor a => NamedColor a where
  type ColorName a :: Symbol
```

`ColorName` is an *associated type family* that let's us map a type of to another type (of kind `Symbol`).

Let's use `NamedColor` to write a function that will let us get the name of a given color as a runtime `String`:

```
colorNameVal :: forall a. KnownSymbol (ColorName a) => String
colorNameVal = symbolVal $ Proxy @(ColorName a)
```

Simple NamedColor Instances

Creating a named color for the colors in our color palet is easy.

```
instance NamedColor RebeccaPurple where
  type ColorName RebeccaPurple = "RebeccaPurple"
```

Simple NamedColor Instances

Creating a named color for the colors in our color palet is easy.

```
instance NamedColor RebeccaPurple where
  type ColorName RebeccaPurple = "RebeccaPurple"
```

We can also create a new type, NamedRGB, that makes it easy for us to give a name to some particular RGB color:

```
data NamedRGB (name :: Symbol) (r :: Nat) (g :: Nat) (b :: Nat) = NamedRGB

instance ValidRGB r g b => IsColor (NamedRGB name r g b) where
  toRGB _ = toRGB $ (RGBColor :: RGBColor r g b)

instance IsColor (NamedRGB name r g b)
  => NamedColor (NamedRGB name r g b) where

  type ColorName _ = name
```

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

```
 colorNameVal @(RGBColor 0 190 239)
"#00BEEF"
```

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

```
 colorNameVal @(RGBColor 0 190 239)
"#00BEEF"
```

From our demo we can see that there are a few key requirements:

- Convert a `Nat` to a `Symbol`, representing it's hex value

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

```
 colorNameVal @(RGBColor 0 190 239)
"#00BEEF"
```

From our demo we can see that there are a few key requirements:

- Convert a `Nat` to a `Symbol`, representing it's hex value
- Pad a hex string out to 2 digits

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

```
 colorNameVal @(RGBColor 0 190 239)
"#00BEEF"
```

From our demo we can see that there are a few key requirements:

- Convert a `Nat` to a `Symbol`, representing it's hex value
- Pad a hex string out to 2 digits
- Append several symbols together to make a human-readable hex string

Renaming Things Is An Even Harder Problem

How can we create a `NamedColor` instance for `RGBColor`? Let's start by looking at a demo of what we want:

```
 colorNameVal @(RGBColor 0 190 239)
"#00BEEF"
```

From our demo we can see that there are a few key requirements:

- Convert a `Nat` to a `Symbol`, representing it's hex value
- Pad a hex string out to 2 digits
- Append several symbols together to make a human-readable hex string
- Use the symbol in our `NamedColor` instance

Hexing The Type Families

We want to convert a `Nat` to a `Symbol`. Whenever we want to create a "function" from one type to another, we can think of type families.

In this case, we'll create a *closed type family* to handle the mapping.

Casting Hexes

```
type family NatHex (n :: Nat) :: Symbol where
  NatHex 0  = "0"
  NatHex 1  = "1"
  NatHex 2  = "2"
  NatHex 3  = "3"
  NatHex 4  = "4"
  NatHex 5  = "5"
  NatHex 6  = "6"
  NatHex 7  = "7"
  NatHex 8  = "8"
  NatHex 9  = "9"
  NatHex 10 = "A"
  NatHex 11 = "B"
  NatHex 12 = "C"
  NatHex 13 = "D"
  NatHex 14 = "E"
  NatHex 15 = "F"
  NatHex n = NatHex (Div n 16) `AppendSymbol` NatHex (Mod n 16)
```

Padding, Part 1

The algorithm to pad a hex string to two digits is pretty straightforward:

Padding, Part 1

The algorithm to pad a hex string to two digits is pretty straightforward:

```
if the number is less than 15:
  return "0" prepended to the stringified value
else:
  return the stringified value
```

It's Looking Kind of Iffy

And so, to left-pad our string, we just need to implement conditional logic at the type level.

It's Looking Kind of Iffy

And so, to left-pad our string, we just need to implement conditional logic at the type level.

```
type family IfThenElse (p :: Bool) (t :: a) (f :: a) where
  IfThenElse True t f = t
  IfThenElse False t f = f
```

Bringing It All Together

Now that we can use conditionals at the type level, we can create a type family to create zero-padded hex strings from naturals:

Bringing It All Together

Now that we can use conditionals at the type level, we can create a type family to create zero-padded hex strings from naturals:

```
type family PadNatHex (n :: Nat) :: Symbol where
  PadNatHex n =
    IfThenElse (n <=? 15) ("0" `AppendSymbol` NatHex n) (NatHex n)
```

Bringing It All Together

Now that we can use conditionals at the type level, we can create a type family to create zero-padded hex strings from naturals:

```
type family PadNatHex (n :: Nat) :: Symbol where
  PadNatHex n =
    IfThenElse (n <=? 15) ("0" `AppendSymbol` NatHex n) (NatHex n)
```

Using `PadNatHex` we can now also write an instance of `NamedColor` for plain `RGBColor` types:

Bringing It All Together

Now that we can use conditionals at the type level, we can create a type family to create zero-padded hex strings from naturals:

```
type family PadNatHex (n :: Nat) :: Symbol where
  PadNatHex n =
    IfThenElse (n <=? 15) ("0" `AppendSymbol` NatHex n) (NatHex n)
```

Using `PadNatHex` we can now also write an instance of `NamedColor` for plain `RGBColor` types:

```
instance IsColor (RGBColor r g b) => NamedColor (RGBColor r g b) where
  type ColorName _ =
    (("#" `AppendSymbol` PadNatHex r)
      `AppendSymbol` PadNatHex g
    ) `AppendSymbol` PadNatHex b
```

Constructing A Theme Instance With `NamedColor`

What's Old Is New Again

Now that we can refer to colors and their names at the type level, we can return to the problem of creating a theme instance whose runtime values are guaranteed to match the theme type parameter.

Constructive Theming

We need a way to create a new `ThemeInstance` with a type parameter that matches the colors we've added to the instance.

Constructive Theming

We need a way to create a new `ThemeInstance` with a type parameter that matches the colors we've added to the instance.

This seems like a job for GADTs

MkTheme

Instead of directly trying to build a `ThemeInstance` we'll have better luck building up an expression and evaluating that to an appropriately typed `ThemeInstance`.

MkTheme

Instead of directly trying to build a `ThemeInstance` we'll have better luck building up an expression and evaluating that to an appropriately typed `ThemeInstance`.

```
data MkTheme theme where
  NewTheme :: MkTheme '[]
  AddColor :: (KnownSymbol (ColorName color), NamedColor color)
           => color
           -> MkTheme theme
           -> MkTheme (ColorName color : theme)
```

MkTheme-ing A ThemeInstance

Once we have constructed a `MkTheme` value that represents our theme, we need to convert it into a `ThemeInstance`:

MkTheme-ing A ThemeInstance

Once we have constructed a `MkTheme` value that represents our theme, we need to convert it into a `ThemeInstance`:

```
instantiateTheme :: MkTheme theme -> ThemeInstance theme
```

MkTheme-ing A ThemeInstance

Once we have constructed a `MkTheme` value that represents our theme, we need to convert it into a `ThemeInstance`:

```
instantiateTheme :: MkTheme theme -> ThemeInstance theme
instantiateTheme NewTheme = ThemeInstance Map.empty
```

MkTheme-ing A ThemeInstance

Once we have constructed a `MkTheme` value that represents our theme, we need to convert it into a `ThemeInstance`:

```
instantiateTheme :: MkTheme theme -> ThemeInstance theme
instantiateTheme NewTheme = ThemeInstance Map.empty
instantiateTheme (AddColor color mkTheme') =
  let
    (ThemeInstance t) = instantiateTheme mkTheme'
    colorName = colorNameVal' color
    colorVal = SomeColor $ toRGB color
  in ThemeInstance $ Map.insert colorName colorVal t
```

A MkTheme Demo

```
sampleColorSet =
  instantiateTheme $
  AddColor (namedRGB @"red"   @255 @0   @0)   $
  AddColor (namedRGB @"green" @0   @255 @0)   $
  AddColor (namedRGB @"blue"  @0   @0   @255) $
  NewTheme

sampleThemer theme = show
  ( lookupColor @"red" theme
  , lookupColor @"green" theme
  , lookupColor @"blue" theme)
```

A MkTheme Demo

```haskell
sampleColorSet =
  instantiateTheme $
  AddColor (namedRGB @"red"   @255 @0   @0)   $
  AddColor (namedRGB @"green" @0   @255 @0)   $
  AddColor (namedRGB @"blue"  @0   @0   @255) $
  NewTheme

sampleThemer theme = show
  ( lookupColor @"red" theme
  , lookupColor @"green" theme
  , lookupColor @"blue" theme)
```

If we run our function in ghci we can see that we're able to get the colors as we expected.

```
> sampleThemer sampleColorSet
(RGB {rgbRed = 255, rgbGreen = 0,   rgbBlue = 0}
,RGB {rgbRed = 0,   rgbGreen = 255, rgbBlue = 0},
,RGB {rgbRed = 0,   rgbGreen = 0,   rgbBlue = 255})
```

Generating A Runtime Theme Configuration

Runtime Configuration

We now have a theming system that works if it compiles...

Runtime Configuration

We now have a theming system that works if it compiles. . .

. . . But we have to compile the application every time we want to make a change.

Type Safety + Runtime Theme Configuration

We can't use the type system to ensure that we always get a correct and complete runtime theme configuration, but we *can* keep our typesafe theme implementation and provide an additional less safe way to generate themes at runtime.

theme.json

We'll start by defining a theme file format. For simplicity, we'll use JSON.

theme.json

We'll start by defining a theme file format. For simplicity, we'll use JSON.

- Top-level keys are theme element names
- Top-level values hold colors
- A color can be an RGB color, an X11 color name

theme.json

We'll start by defining a theme file format. For simplicity, we'll use JSON.

- Top-level keys are theme element names
- Top-level values hold colors
- A color can be an RGB color, an X11 color name
- ... Or *a reference to another color*

theme.json

We'll start by defining a theme file format. For simplicity, we'll use JSON.

- Top-level keys are theme element names
- Top-level values hold colors
- A color can be an RGB color, an X11 color name
- ... Or *a reference to another color*

```
{
    "red": {"rgb": "#ff0000"},
    "green": {"rgb": "#00ff00"},
    "blue": {"x11": "AliceBlue"},
    "text": {"same-as": "blue"},
    "border": {"same-as": "text"}
}
```

Defining The Theme Config Format

A theme that's configured at runtime can be represented as a map from strings to some sort of a color value:

```
newtype ThemeConfig = ThemeConfig
  {getThemeConfig :: Map.Map String ColorValue}
```

But how do we represent the individual color values?

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors
- References to other colors

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors
- References to other colors

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors
- References to other colors

The first two cases are pretty easy to handle:

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors
- References to other colors

The first two cases are pretty easy to handle:

```
data ColorValue
  = RGBValue RGB
  | X11Value SomeColor
```

Creating A Runtime Color Value

We have three different types of values we can use for an element in our theme:

- RGB colors
- Named X11 Colors
- References to other colors

The first two cases are pretty easy to handle:

```
data ColorValue
  = RGBValue RGB
  | X11Value SomeColor
```

What about references?

Reader References

An idiomatic way to handle references to other part of our data structure would be to use a `Reader`.

Reader References

An idiomatic way to handle references to other part of our data structure would be to use a `Reader`.

```
newtype ColorReference r a =
  ColorReference {unColorReference :: ExceptT String (Reader r) a}
  deriving newtype (Functor, Applicative, Monad, MonadReader r, MonadError Stri

  data ColorValue
    = RGBValue RGB
    | X11Value SomeColor
    | OtherColor (ColorReference ThemeConfig ColorValue)
```

The Reader Problem

But we have some problems here:

- Every reader is a now a landmine that could fail when we read it

The Reader Problem

But we have some problems here:

- Every reader is a now a landmine that could fail when we read it
- We won't even know right away if something has failed- only when (if) we actually use that value

The Reader Problem

But we have some problems here:

- Every reader is a now a landmine that could fail when we read it
- We won't even know right away if something has failed- only when (if) we actually use that value
- Every time we want to access a color element, we have to run the reader, reducing ergonomics

Strictly Evaluating A Color Value

One way we could handle some of the problems with our color value is to introduce a function that strictly evaluates a `ColorValue`.
We'll start by introducing another type:

Strictly Evaluating A Color Value

One way we could handle some of the problems with our color value is to introduce a function that strictly evaluates a `ColorValue`.
We'll start by introducing another type:

```
data StrictColorValue
  = StrictRGBValue RGB
  | StrictX11Value SomeColor
  | StrictOtherColor StrictColorValue
```

Strictly Evaluating A Color Value

One way we could handle some of the problems with our color value is to introduce a function that strictly evaluates a `ColorValue`.
We'll start by introducing another type:

```
data StrictColorValue
  = StrictRGBValue RGB
  | StrictX11Value SomeColor
  | StrictOtherColor StrictColorValue
```

Now we can write a function to handle the conversion strictly, ensuring that we catch any failures upfront:

Strictly Evaluating A Color Value

One way we could handle some of the problems with our color value is to introduce a function that strictly evaluates a `ColorValue`.
We'll start by introducing another type:

```
data StrictColorValue
  = StrictRGBValue RGB
  | StrictX11Value SomeColor
  | StrictOtherColor StrictColorValue
```

Now we can write a function to handle the conversion strictly, ensuring that we catch any failures upfront:

```
strictlyEvaluateColorValue :: ColorValue -> Either String StrictColorValue
```

Custom Reader References

Strictly evaluating our reference solves both the problem of hidden errors and makes accessing colors more ergonomic, but now we've introduced a new problem: We have two copies of our data structure, and we have to maintain both of them.

Custom Reader References

Strictly evaluating our reference solves both the problem of hidden errors and makes accessing colors more ergonomic, but now we've introduced a new problem: We have two copies of our data structure, and we have to maintain both of them. We can address this problem with a little help

from type families and the *Higher-Kinded Data* pattern.

The Higher-Kinded Data Pattern

The higher-kinded data pattern lets us take a higher-kinded type and erase it if it's `Identity`:

```
type family HKD (wrapper :: Type -> Type) (value :: Type) :: Type where
  HKD Identity value = value
  HKD wrapper value = wrapper value
```

The Higher-Kinded Data Pattern

The higher-kinded data pattern lets us take a higher-kinded type and erase it if it's `Identity`:

```
type family HKD (wrapper :: Type -> Type) (value :: Type) :: Type where
  HKD Identity value = value
  HKD wrapper value = wrapper value
```

We can see how this works in practice in GHCI:

The Higher-Kinded Data Pattern

The higher-kinded data pattern lets us take a higher-kinded type and erase it if it's `Identity`:

```
type family HKD (wrapper :: Type -> Type) (value :: Type) :: Type where
  HKD Identity value = value
  HKD wrapper value = wrapper value
```

We can see how this works in practice in GHCI:

```
> :kind! HKD (ColorReference RawThemeConfig) ColorValue
HKD (ColorReference RawThemeConfig) ColorValue :: *
= ColorReference RawThemeConfig ColorValue

> :kind! HKD Identity ColorValue
HKD Identity ColorValue :: *
= ColorValue
```

Using Higher-Kinded Data Improve `ColorValue`

We can use higher-kinded data now to wrap our reference and have a single `ColorValue` type:

Using Higher-Kinded Data Improve `ColorValue`

We can use higher-kinded data now to wrap our reference and have a single `ColorValue` type:

```haskell
data ColorValue w
  = RGBValue RGB
  | X11Value SomeColor
  | OtherColor (HKD w (ColorValue w))
```

Updating `ThemeConfig`

The next step is we need to refactor `ThemeConfig` to pass the right type into `ColorValue`. Let's make that a parameter to `ThemeConfig`:

Updating ThemeConfig

The next step is we need to refactor `ThemeConfig` to pass the right type into `ColorValue`. Let's make that a parameter to `ThemeConfig`:

```
newtype ThemeConfig w = ThemeConfig
  {getThemeConfig :: Map.Map String (ColorValue w)}
```

Updating ThemeConfig

The next step is we need to refactor `ThemeConfig` to pass the right type into `ColorValue`. Let's make that a parameter to `ThemeConfig`:

```
newtype ThemeConfig w = ThemeConfig
  {getThemeConfig :: Map.Map String (ColorValue w)}
```

But what should `w` be?

Updating ThemeConfig

The next step is we need to refactor `ThemeConfig` to pass the right type into `ColorValue`. Let's make that a parameter to `ThemeConfig`:

```
newtype ThemeConfig w = ThemeConfig
  {getThemeConfig :: Map.Map String (ColorValue w)}
```

But what should `w` be?
We can't directly pass a `ColoreReference` that holds a `ThemeConfig` because the type would be infinite:

Updating ThemeConfig

The next step is we need to refactor ThemeConfig to pass the right type into ColorValue. Let's make that a parameter to ThemeConfig:

```
newtype ThemeConfig w = ThemeConfig
  {getThemeConfig :: Map.Map String (ColorValue w)}
```

But what should w be?
We can't directly pass a ColoreReference that holds a ThemeConfig because the type would be infinite:

```
ColorValue (ColoreReference (ThemeConfig (ColorReference (ThemeConfig ...
```

Wrapping Up The Theme Config

Thankfully we have an easy way out of this problem using a newtype. First, let's rename `ThemeConfig` and create a type alias for a resolved configuration:

Wrapping Up The Theme Config

Thankfully we have an easy way out of this problem using a newtype. First, let's rename `ThemeConfig` and create a type alias for a resolved configuration:

```
newtype ThemeConfig' w = ThemeConfig'
  {getThemeConfig :: Map.Map String (ColorValue w)}

type ThemeConfig = ThemeConfig' Identity
```

Wrapping Up The Theme Config

Thankfully we have an easy way out of this problem using a newtype. First, let's rename `ThemeConfig` and create a type alias for a resolved configuration:

```haskell
newtype ThemeConfig' w = ThemeConfig'
  {getThemeConfig :: Map.Map String (ColorValue w)}

type ThemeConfig = ThemeConfig' Identity
```

Next, we'll add a *newtype wrapper* that doesn't take a type parameter. This is what we'll reference:

Wrapping Up The Theme Config

Thankfully we have an easy way out of this problem using a newtype. First, let's rename `ThemeConfig` and create a type alias for a resolved configuration:

```
newtype ThemeConfig' w = ThemeConfig'
  {getThemeConfig :: Map.Map String (ColorValue w)}

type ThemeConfig = ThemeConfig' Identity
```

Next, we'll add a *newtype wrapper* that doesn't take a type parameter. This is what we'll reference:

```
newtype RawThemeConfig = RawThemeConfig
  { getRawThemeConfig :: ThemeConfig' (ColorReference RawThemeConfig) }
```

Evaluating The ThemeConfig

As a final step, we can implement our strict evaluation code and return a clean and valid theme configuration:

```
evalConfig :: RawThemeConfig -> Either String ThemeConfig
```

ThemeConfig Demo

```
> loadThemeConfig "../theme.json"
ThemeConfig'
  {getThemeConfig = fromList
    [ ("blue",X11Value RGB {rgbRed = 240, rgbGreen = 248, rgbBlue = 255})
    , ("border",OtherColor (OtherColor (X11Value RGB {rgbRed = 240, rgbGreen =
    , ("green",RGBValue (RGB {rgbRed = 0, rgbGreen = 255, rgbBlue = 0}))
    , ("red",RGBValue (RGB {rgbRed = 255, rgbGreen = 0, rgbBlue = 0}))
    , ("text",OtherColor (X11Value RGB {rgbRed = 240, rgbGreen = 248, rgbBlue =
```

Integrating The Runtime and Type Level Theming Systems

Bringing It All Together

Now we have a way to handle queries safely at the type level using a `ThemeInstance` and we have a way to get a theme configuration at runtime with `ThemeConfig`. How can we bring these two together?

Theme Validation

To validate that our runtime theme matches the type-level theme definition, we need to step through each element of our theme type and ensure that it exists in the configured theme. We'll start with a new typeclass:

```
class ValidateThemeInstance (theme :: Theme) (a :: Theme -> Type) where
  validateThemeInstance :: Map.Map String SomeColor -> Either String (a theme)
```

Theme Validation

To validate that our runtime theme matches the type-level theme definition, we need to step through each element of our theme type and ensure that it exists in the configured theme. We'll start with a new typeclass:

```
class ValidateThemeInstance (theme :: Theme) (a :: Theme -> Type) where
  validateThemeInstance :: Map.Map String SomeColor -> Either String (a theme)
```

And an instance for our base case, an empty theme

Theme Validation

To validate that our runtime theme matches the type-level theme definition, we need to step through each element of our theme type and ensure that it exists in the configured theme. We'll start with a new typeclass:

```
class ValidateThemeInstance (theme :: Theme) (a :: Theme -> Type) where
  validateThemeInstance :: Map.Map String SomeColor -> Either String (a theme)
```

And an instance for our base case, an empty theme

```
instance ValidateThemeInstance '[] ThemeInstance where
  validateThemeInstance theme = Right (ThemeInstance theme)
```

Stepping Through Theme Validation

When we have a non-empty theme, our validation algorithm is fairly straightforward: Get the runtime representation of the element at the head of our theme list. If it's in the runtime map, add it to the theme instance and recurse. Otherwise fail.

Stepping Through Theme Validation

When we have a non-empty theme, our validation algorithm is fairly straightforward: Get the runtime representation of the element at the head of our theme list. If it's in the runtime map, add it to the theme instance and recurse. Otherwise fail.

```
instance ( KnownSymbol currentColor
         , ValidateThemeInstance rest ThemeInstance
         ) => ValidateThemeInstance (currentColor:rest) ThemeInstance where
  validateThemeInstance theme =
    let targetColor = symbolVal $ Proxy @currentColor
    in case Map.lookup targetColor theme of
      Nothing ->
        let colorName = symbolVal $ Proxy @currentColor
        in Left $ "missing color: " <> colorName
      Just _ -> do
        (ThemeInstance m) <- validateThemeInstance @rest theme
        pure $ ThemeInstance m
```

Runtime Theme Config Demo

Let's look at how we can use this in a real program

```
type RuntimeTheme = ["blue", "green", "red"]
validateThemeConfig
  :: forall (theme :: Theme).
     ValidateThemeInstance theme ThemeInstance
  => ThemeConfig
  -> Either String (ThemeInstance theme)
validateThemeConfig =
  validateThemeInstance . Map.map SomeColor . getThemeConfig

testQuery :: FilePath -> IO ()
testQuery p = do
  cfg <- loadThemeConfig p
  let
    sampleQuery t = (lookupColor @"red" t, lookupColor @"blue" t)
    r = sampleQuery <$> validateThemeConfig @RuntimeTheme cfg
  print r
```

Questions