

Make It Purple

An Introduction to Type Level Programming

Rebecca Skinner

November 7, 2022

Prelude

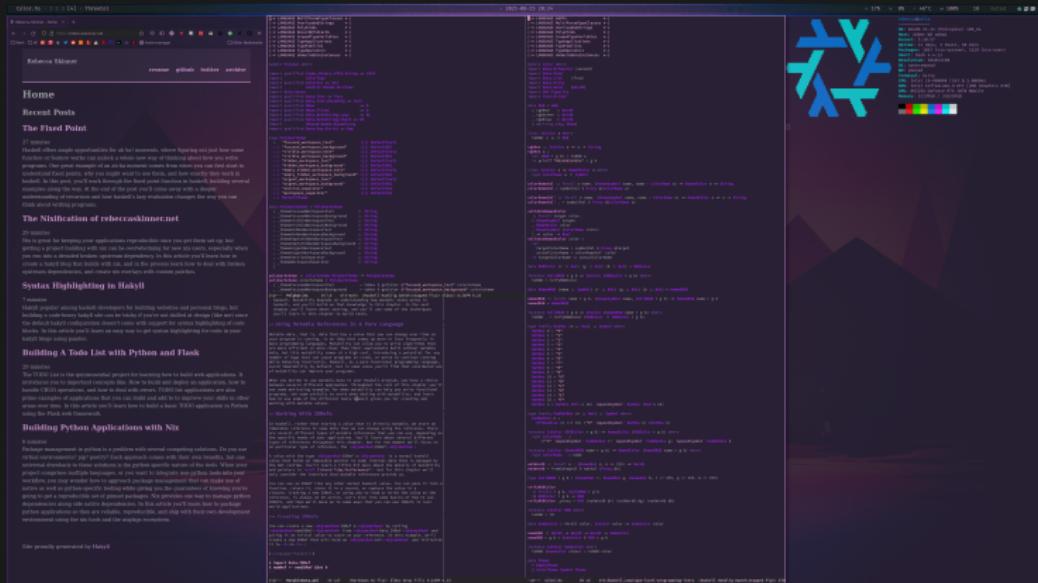
Hello, World

- ▶ 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      = "#ddaa00"
, focusedWorkspaceBackground = "#2a2035"
, visibleWorkspaceText      = "#ddaa00"
, visibleWorkspaceBackground = "#2a2035"
}
```

Let's Build It!

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

```
polybarColorScheme = PolybarColorScheme
{ focusedWorkspaceText      = "#ddaa00"
, focusedWorkspaceBackground = "#2a2035"
, visibleWorkspaceText      = "#ddaa00"
, visibleWorkspaceBackground = "#2a2035"
}
```

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

```
stylishConfig :: XConfig a -> XConfig a
stylishConfig cfg = cfg
{ normalBorderColor  = "#ddaa00"
, 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*.

```
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 belong to the theme somehow.

A Theme Instance With Phantom Types

We need to keep track of all of the theme elements that belong 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 belong 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

Let's make a theme instance!

A Phantom Type With a Theme Kind

Let's make a theme instance!

```
myTheme :: ThemeInstance (Int, Either String Bool)
myTheme = ThemeInstance Map.empty
```

A Phantom Type With a Theme Kind

Let's make a theme instance!

```
myTheme :: ThemeInstance (Int, Either String Bool)
myTheme = ThemeInstance Map.empty
```

No...

A Phantom Type With a Theme Kind

Let's make a theme instance!

```
myTheme :: ThemeInstance (Int, Either String Bool)
myTheme = ThemeInstance Map.empty
```

No...

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

Let's make a theme instance!

```
myTheme :: ThemeInstance (Int, Either String Bool)
myTheme = ThemeInstance Map.empty
```

No...

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

A Symbol is a type-level String.

Symbolism

A Symbol is a type-level String.

```
> :kind "green"
"green" :: Symbol
```

Symbolism

A Symbol is a type-level String.

```
> :kind "green"
"green" :: Symbol
```

KnownSymbol is a typeclass that let's us get a string from a symbol.

```
> symbolVal $ Proxy @"green"
"green"
> :t symbolVal
symbolVal :: KnownSymbol n => proxy n -> String
```

A Theme Of Many Colors

Great. But we want a *Theme* not a single color.

A Theme Of Many Colors

Great. But we want a *Theme* not a single color.

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

Checking The Colors In A Theme

We have a list of colors now. But is it the right list?

Checking The Colors In A Theme

We have a list of colors now. But is it the right list?

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

Implementing HasColor

The secret is in HasColor

Implementing HasColor

The secret is in HasColor

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

Implementing HasColor

The secret is in HasColor

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

To make HasColor work, we need recursion. We'll start with a base case:

Implementing HasColor

The secret is in HasColor

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

To make HasColor work, we need recursion. We'll start with a base case:

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

Implementing HasColor

The secret is in HasColor

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

To make HasColor work, we need recursion. We'll start with a base case:

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

We'll call the recursive case if the head of the theme isn't what we want.

Implementing HasColor

The secret is in HasColor

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

To make HasColor work, we need recursion. We'll start with a base case:

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

We'll call the recursive case if the head of the theme isn't what we want.

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

Dealing with Overlapping Instances

And now we have a problem.

- ▶ `color` and `color` are always the same.

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

Dealing with Overlapping Instances

And now we have a problem.

- ▶ `color` and `color` are always the same.
- ▶ `color` and `currentColor` might be the same.

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

Getting A Theme Element

HasColor Accomplished. Let's get colorful.

Getting A Theme Element

HasColor Accomplished. Let's get colorful.

```
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

```
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)]
```

We succeed when we should.

```
> 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)]
```

We succeed when we should.

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

More importantly, we fail when we should.

```
> 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

```
> :t demoThemeInstance
demoThemeInstance :: ThemeInstance ["red","green","blue"]

> 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
```

Mind The Gap

```
> :t demoThemeInstance
demoThemeInstance :: ThemeInstance ["red","green","blue"]

> 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
```

But we don't always fail when we should

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)]
```

Fixing The Problem

We need to construct the theme and the value at the same time.

Colors At The Type Level: Part 2

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

X11 Colors At The Type Level

We can build a type-level color pallet with X11 colors.

```
data RebeccaPurple = RebeccaPurple
```

X11 Colors At The Type Level

We can build a type-level color pallet with X11 colors.

```
data RebeccaPurple = RebeccaPurple
```

But that's a lot of typing

```
user@host$ wc -l ColorX11.hs
1488 ColorX11.hs
```

X11 Colors At The Type Level

We can build a type-level color pallet with X11 colors.

```
data RebeccaPurple = RebeccaPurple
```

But that's a lot of typing

```
user@host$ wc -l ColorX11.hs
1488 ColorX11.hs
```

And we're still limited to a fixed palette of colors.

Type Level RGB

What if RGB, but at the type level?

Type Level RGB

What if RGB, but at the type level?

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

Type Level RGB Is A Color

We still need to get a runtime representation of color.

Type Level RGB Is A Color

We still need to get a runtime representation of color.

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

instance IsColor RGB where
    toRGB = id

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

The RGB Problem

```
toRGB (RGBColor @975 @2148 @8)
```

Making an RGBColor Type Into a Runtime Value

Ok, how about only valid RGB Colors are colors?

Making an RGBColor Type Into a Runtime Value

Ok, how about only valid RGB Colors are colors?

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 RGB Value

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

The RGBColor Problem

```
polybarColorScheme = PolybarColorScheme
{ focusedWorkspaceText      = RGBColor :: RGBColor 255 255 255
, focusedWorkspaceBackground = RGBColor :: RGBColor 0 0 0
--- more like this
```

The RGBColor Problem

```
polybarColorScheme = PolybarColorScheme
{ focusedWorkspaceText      = RGBColor :: RGBColor 255 255 255
, focusedWorkspaceBackground = RGBColor :: RGBColor 0 0 0
--- more like this
```

Remind you of anything?

The RGBColor Problem

```
polybarColorScheme = PolybarColorScheme
{ focusedWorkspaceText      = RGBColor :: RGBColor 255 255 255
, focusedWorkspaceBackground = RGBColor :: RGBColor 0 0 0
--- more like this
```

Remind you of anything?

```
polybarColorScheme = PolybarColorScheme
{ focusedWorkspaceText      = "#ddaa00"
, focusedWorkspaceBackground = "#2a2035"
```

The Hardest Problem in Computer Science

The next in a series of problems

- We've got type level names for colors: Symbol

The Hardest Problem in Computer Science

The next in a series of problems

- ▶ We've got type level names for colors: Symbol
- ▶ We've got type level colors: RGBColor

The Hardest Problem in Computer Science

The next in a series of problems

- ▶ We've got type level names for colors: Symbol
- ▶ We've got type level colors: RGBColor
- ▶ We've got no way to connect them together

Type Families

We need a function from `RGBColor r g b` to `Symbol`

Type Families

We need a function from `RGBColor r g b` to `Symbol` A function between types is called a **Type Family**

Creating A Type Family For Color Names

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

ColorName is an *associated type family*

Creating A Type Family For Color Names

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

ColorName is an *associated type family*

```
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"
```

When In Doubt, Hardcode It

To name an RGBColor, we just give it a name:

When In Doubt, Hardcode It

To name an RGBColor, we just give it a name:

```
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

But we really should support naming `RGBColor` too:

Renaming Things Is An Even Harder Problem

But we really should support naming RGBColor too:

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

Hexing The Type Families

NatHex is a *Closed Type Family* from Nat to Symbol

Casting Hexes

```
type family NatHex (n :: Nat) :: Symbol where
  NatHex 0 = "0"
  NatHex 1 = "1"
  NatHex 2 = "2"
  -- And so on
  NatHex 14 = "E"
  NatHex 15 = "F"
  NatHex n = NatHex (Div n 16) `AppendSymbol` NatHex (Mod n 16)
```

Casting Hexes

```
type family NatHex (n :: Nat) :: Symbol where
  NatHex 0  = "0"
  NatHex 1  = "1"
  NatHex 2  = "2"
  -- And so on
  NatHex 14 = "E"
  NatHex 15 = "F"
  NatHex n = NatHex (Div n 16) `AppendSymbol` NatHex (Mod n 16)
```

```
> :kind! NatHex 11
NatHex 11 :: Symbol
= "B"
> :kind! NatHex 250
NatHex 250 :: Symbol
= "FA"
```

Padding, Part 1

Let's pad those numbers

```
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

If it were only so easy...

It's Looking Kind of Iffy

If it were only so easy...

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

Bringing It All Together

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

Bringing It All Together

```
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

```
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

Let's make a well-typed theme at runtime.

What's Old Is New Again

Let's make a well-typed theme at runtime. A runtime function needs to
be called with *values*

What's Old Is New Again

Let's make a well-typed theme at runtime. A runtime function needs to
be called with *values* So we need A Function from Values to Types.

MkTheme

A **GADT** is a function from a value to a type

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

MkTheme-ing A ThemeInstance

What we build up, we must tear down.

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

MkTheme-ing A ThemeInstance

What we build up, we must tear down.

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

MkTheme-ing A ThemeInstance

What we build up, we must tear down.

```
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

Let's try it out:

A MkTheme Demo

Let's try it out:

```
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

Let's try it out:

```
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)
```

```
> 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

Users don't like to compile things. Let's make it work at runtime too.

Runtime Configuration

Users don't like to compile things. Let's make it work at runtime too.

At the cost of some safety.

theme.json

At least it's not YAML...

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

Defining The Theme Config Format

Another dictionary will solve things

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

Creating A Runtime Color Value

Sometimes things are easy

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

Creating A Runtime Color Value

Sometimes things are easy

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

And sometimes they are hard: What about references?

Reader References

Let's use monad transformers for some reason.

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

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

The Reader Problem

This kind of sucks.

- ▶ Using reader might fail

The Reader Problem

This kind of sucks.

- ▶ Using reader might fail
- ▶ And it might not fail immediately

The Reader Problem

This kind of sucks.

- ▶ Using reader might fail
- ▶ And it might not fail immediately
- ▶ And it's not very ergonomic

Strictly Evaluating A Color Value

Is this *Strict Haskell* ?

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

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

Custom Reader References

Let's try it again without gratuitous duplication.

The Higher-Kinded Data Pattern

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

Using Higher-Kinded Data Improve ColorValue

Using HKD is easy, if a little odd looking.

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

Updating ThemeConfig

Better parameterize ThemeConfig too or we've just moved the problem around.

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

Infinite Failure

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

If a ColorValue References a theme config, the type of w must be:

Infinite Failure

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

If a ColorValue References a theme config, the type of w must be:

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

Wrapping Up The Theme Config

This is a problem we *can* solve by just not looking at it.

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

type ThemeConfig = ThemeConfig' Identity

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

Evaluating The ThemeConfig

Strictly evaluating our references is looking a lot better now

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

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

- ▶ We have typesafe theme configuration.
- ▶ And a way to load a theme at runtime.
- ▶ Again, let's make these work together.

Theme Validation

Validation is a function from a type-level Theme to a value-level ThemeInstance

Functions from types to values are *type classes* so let's start there.

Theme Validation

Validation is a function from a type-level Theme to a value-level ThemeInstance

Functions from types to values are *type classes* so let's start there.

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

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

Stepping Through Theme Validation

```
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