`:kind` of a Pointless* Talk

It's not about tacit programming!

Jan Sliacky

Terminology

Value-level Terms (Expressions)

Types

Kinds

Expressions

```
23

“Hello World!”

23 + 42

True && False
```

Types

```
> 23 :: Int

> “Hello World!” :: String

> 23 + 42 :: Int (assuming + :: Int → Int → Int)

> True && False :: Bool (assuming && :: Bool → Bool → Bool)
```

Kinds

Types of Types

Kind * aka Type

Types of kind * can have values.

Kinds of Simple Types

```
> Int :: *

> String :: *

> Int → Int → Int :: *
```

Kinds of Too Simple Types

```
> Int :: *

> String :: *

> Int → Int → Int :: *
```

Custom Data Types

Still :: *

```
data Bool = True | False


data List'of'Ints = Nil
                  | Cons Int List'of'Ints

empty           = Nil
a'few'numbers   = Cons 1 (Cons 2 (Cons 3 Nil))
```

Polymorphic Data Types

Types Abstracting over other Types

```
data List a = Nil
            | Cons a (List a)


data Maybe a = Nothing
             | Just a


data Either a b = Left a
                | Right b
```

Kinds of Polymorphic Data Types

aka Type Constructors

```
> List :: * → *

> Maybe :: * → *

> Either :: * → * → *
```

Higher-kinded Types

Types Abstracting over Types Abstracting over Types

Higher-kinded Types

Types Abstracting over (Types Abstracting over Types)

Higher-kinded Types

```
data Container m a = Contain (m a)
```

```
> :kind Container
```

Higher-kinded Types

```
data Container m a = Contain (m a)
```

```
> :kind Container
> Container :: (* → *) → * → *
```

example

```
data Container m a = Contain (m a)

list'of'ints = Contain (Cons 1 Nil)


> :type list'of'ints

> list'of'ints :: Container List Int
```

Grammar of Kinds

```
Kind k, l = *
          | k → l
```

Kind Polymorphism

Exposing lies (mostly mine)

```
data Container m a = Contain (m a)
```

```
> :kind Container
```

Kind Polymorphism

Exposing lies (mostly mine)

```
data Container m a = Contain (m a)
```

```
> :kind Container

> Container :: (k → *) → k → *
```

Custom Kinds

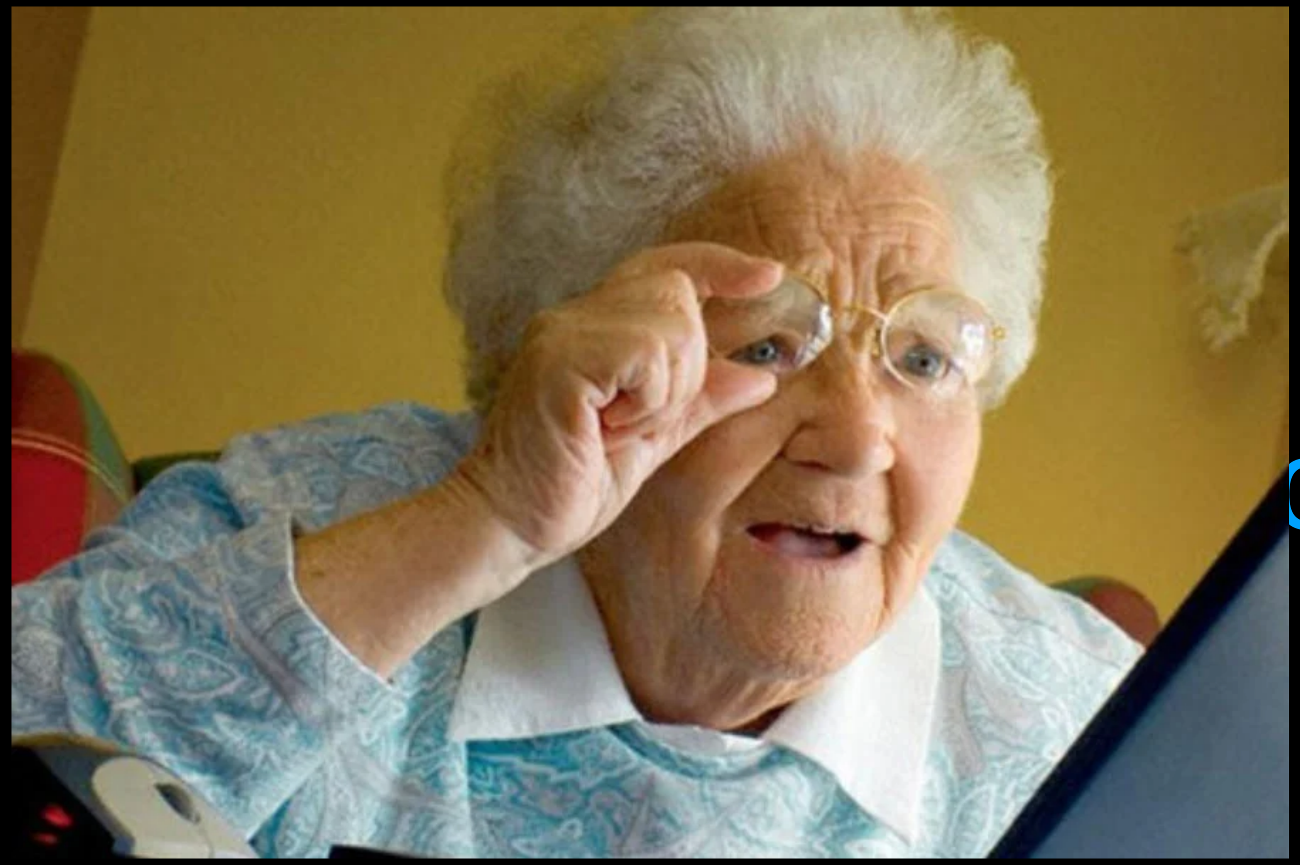
For more kind-level goodness.

```
data Response i = R String

data Valid
data Unknown
```

```haskell
data Response i = R String

data Valid
data Unknown


validate :: Response Unknown → Maybe (Response Valid)
```

```haskell
data Response i = R String

data Valid
data Unknown


validate :: Response Unknown → Maybe (Response Valid)


derp :: Response Bool
```

```
data Response i = R String

data Valid
data Unknown
```



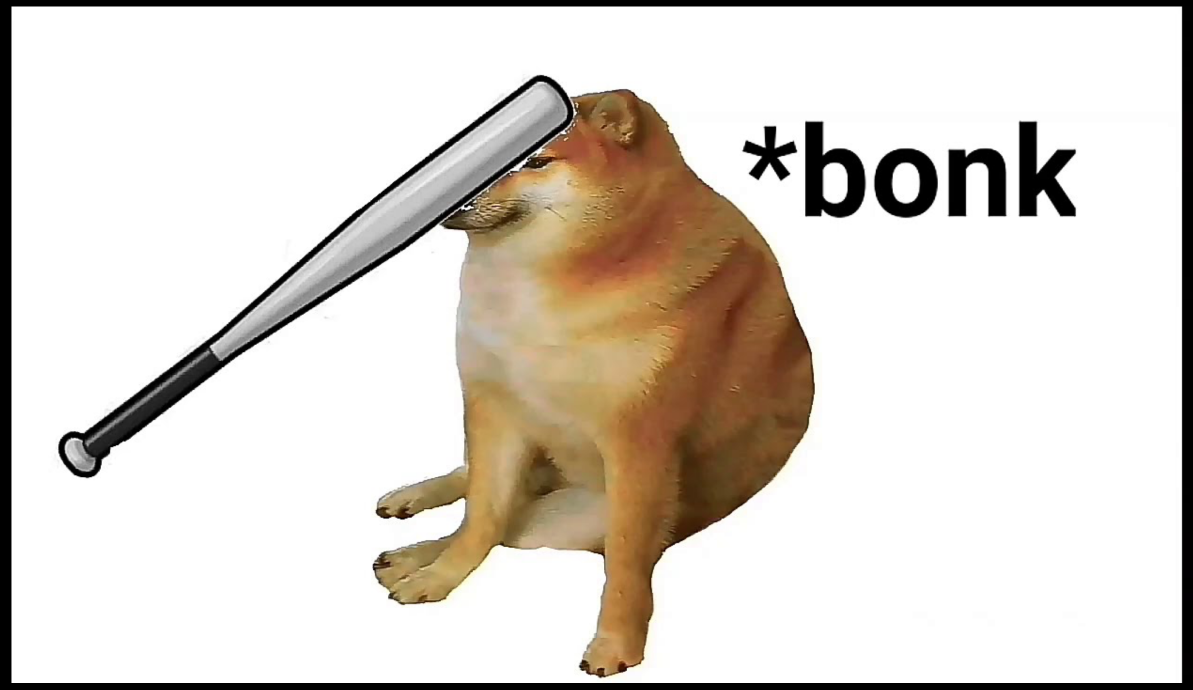
```
onse Unknown → Maybe (Response Valid)

derp :: Response Bool
```

Back to the drawing board

Let’s engage those galaxy brains

```
data Response (i :: Response'I) = R String

kind Response'I

data Valid   :: Response'I
data Unknown :: Response'I

validate :: Response Unknown → Maybe (Response Valid)
```

```haskell
data Response (i :: Response'I) = R String

kind Response'I

data Valid :: Response'I
data Unknown :: Response'I

validate :: Response Unknown → Maybe (Response Valid)

derp :: Response Bool
```



What about Haskell?

Haskell unifies Types and Kinds.

Extensions like `DataKinds` promote types into `kinds` and data constructors into `types`.

Haskell does not provide a facility showed in the previous slides. (It does not need it though.)

Values with Types of Custom Kinds

The what now?

The Fantasy Land part of the Talk.

Kind * aka Type

Types of kind * can have values.

Kind * aka Type

Only types of kind * can have values.

Kind * aka Type

~~Only types of kind * can have values.~~

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val
```

```
kind Foo’Kind

type Foo’Type :: Foo’Kind

data Foo’Type = Foo’Val


some’foo = Foo’Val
```

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val


some'foo = Foo'Val


> :type some'foo
```

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val


some'foo = Foo'Val


> :type some'foo

> some'foo :: Foo'Type
```

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val


id'foo :: ∀ (a :: Foo'Kind) . a → a
id'foo x = x
```

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val

id'foo :: ∀ (a :: Foo'Kind) . a → a
id'foo x = x

> id'foo Foo'Val
> Foo'Val


> id'foo True
> ERROR!
```

```
kind Foo'Kind

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val

id'foo :: ∀ (a :: Foo'Kind) . a → a
id'foo x = x

> id'foo Foo'Val
> Foo'Val


> id'foo True
> ERROR!
```

So what about our ordinary `identity` function?

```
id x = x

> id Foo'Val

> ERROR!
```

Let’s fix that

The real identity™

```
id :: a → a
id x = x
```

Let’s fix that

The real identity™

```
id :: ∀ (a :: *) . a → a
id x = x
```

Let's fix that

The real identity™

```
id :: ∀ (a :: *) . a → a
id x = x
```

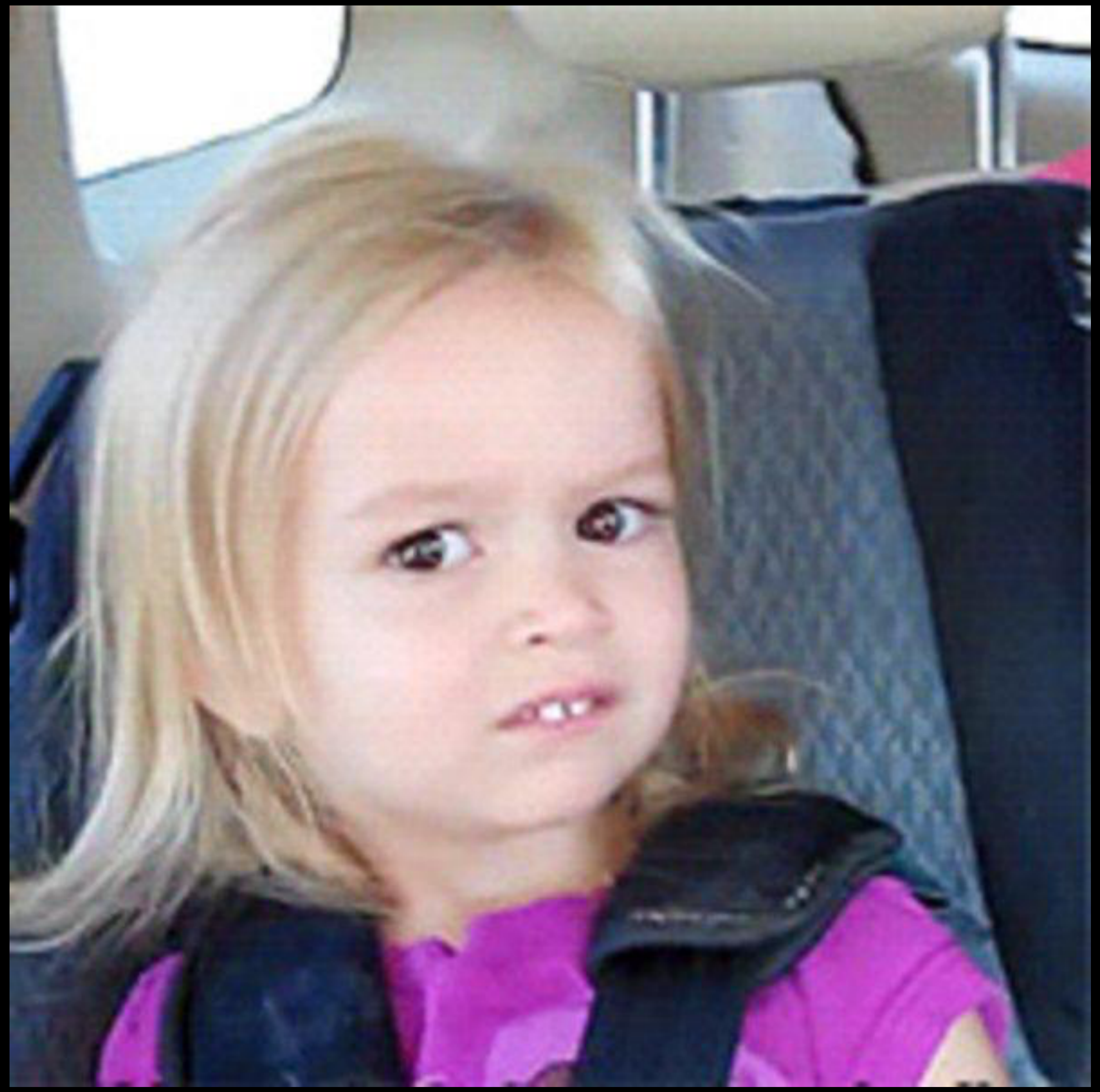
```
real'id :: ∀ (a :: k) . a → a
real'id x = x
```

Let’s fix that

The real identity™

```
id :: ∀ (a :: *) . a → a
id x = x
```

```
real’id :: ∀ (a :: k) . a → a
real’id x = x
```



```
real'id :: ∀ (a :: k) . a → a
real'id x = x
```

```
> real'id Foo'Val
> Foo'Val
```

```
> real'id True
> True
```

Uh, oh

```
data Broken (x :: k) = Break x
```

Uh, oh

```
data Broken (x :: k) = Break x

broken :: Broken Maybe
broken = Break ???
```

Uh, oh

```
data Broken (x :: k) = Break x
```

```
broken :: Broken Maybe
broken = Break ???
```

No wait, we can fix this!

But how?

Maybe with sub-kinding?

Maybe with set-theoretical kind polymorphism?

Sub-kinding?

A ⩽ B means A is a sub-kind of B

```
kind Foo'Kind ⩽ *

type Foo'Type :: Foo'Kind

data Foo'Type = Foo'Val


real'id :: ∀ (a ⩽ *) . a → a
```

Set-theoretical kind polymorphism?

`* | Foo'Kind` is a union of those two kinds.

```
real'id :: ∀ (a :: * | Foo'Kind) . a → a
real'id x = x
```

That is a Different Talk though.

Kind * aka Type

Only types of kind * can have values.

Kind * aka Type

~~Only types of kind * can have values.~~

Kind # in Haskell

Kind for unlifted types.

Also see levity-polymorphism in GHC.

Remember to be kind!

Resources

- https://www.parsonsmatt.org/2017/04/26/basic_type_level_programming_in_haskell.html
- https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/kind-polymorphism.html
- https://wiki.haskell.org/Kind