Values, Types and Kinds

Value
"cat"
3
Just "frog"
\a b -> a + b + 1
[ ]

expressions

Value	Type
"cat"	String
3	Int
Just "frog"	Maybe String
\a b -> a + b + 1	Int -> Int -> Int
[ ]	[a]

expressions

type expressions

Value	Type	Kind
"cat"	String	Type
3	Int	Type
Just "frog"	Maybe String	Type
\a b -> a + b + 1	Int -> Int -> Int	Type
[ ]	[a]	Type

expressions

type expressions

kind expressions

"Type" a.k.a. "*"

Type

The kind of data types.
All values have types of kind *Type*.

e.g.
String :: Type

```
$ ghci
GHCi, version 8.6.0.20180810: http://www.haskell.org/ghc/  :? for help

> :type "cat"
"cat" :: [Char]

> :t Just "frog"
Just "frog" :: Maybe [Char]
```

String = [Char]

```
> :type "cat"
"cat" :: [Char]

> :t Just "frog"
Just "frog" :: Maybe [Char]
```

```
> :kind String
String :: Type

> :k Maybe String
Maybe String :: Type
```

```haskell
data Maybe a = Nothing | Just a
data Maybe a = Nothing | Just a
```

```
a :: Maybe String
a = Just "frogs"

> :type a
a :: Maybe String

> :kind Maybe String
Maybe String :: Type
```

```
x :: Maybe (Maybe String)
x = Just (Just "frogs")

> :type x
x :: Maybe (Maybe String)

> :kind Maybe (Maybe String)
Maybe (Maybe String) :: Type
```

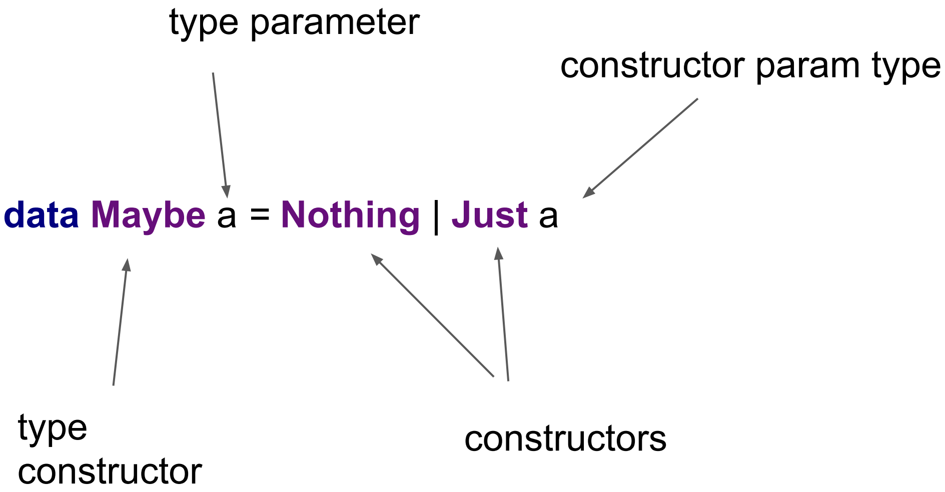
```haskell
data Maybe a = Nothing | Just a

x :: Maybe Maybe
x = undefined
```

```
Frogs2.hs:4:12: error:
     • Expecting one more argument to ‘Maybe’
     Expected a type, but  ‘Maybe’ has kind ‘Type -> Type’
     • In the first argument of ‘Maybe’, namely ‘Maybe’
     In the type signature: x :: Maybe Maybe
   |
4  | x :: Maybe Maybe
   |           ^^^^^
Failed, no modules loaded.
```

```
data Maybe a = Nothing | Just a
```

```
> :kind Maybe
Maybe :: Type -> Type
```



type parameter

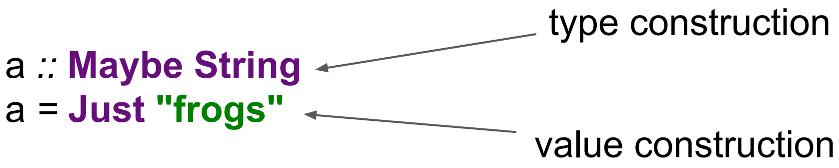
constructor param type

```
data Maybe a = Nothing | Just a
```

type constructor

constructors

```
data Maybe a = Nothing | Just a
```



```
a :: Maybe String
a = Just "frogs"
```

type construction

value construction

expression

Just :: a -> Maybe a ← constructor - function

"frog" :: String

Just "frog" :: Maybe String ← construction - function application

type expression

Maybe :: Type -> Type ← type constructor - type-level function

String :: Type

Maybe String :: Type ← type construction - type-level function application

Type -> Type

The kind of unary type constructors.
Types of this kind have no values.

e.g.
Maybe :: Type -> Type

```haskell
data Maybe a = Nothing | Just a
```

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}

import Data.Kind

data Maybe :: Type -> Type where
 Nothing  :: Maybe a
 Just     :: a -> Maybe a
```

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ExplicitForAll #-}

import Data.Kind

data Maybe :: Type -> Type where
 Nothing  :: forall a. Maybe a
 Just     :: forall a. a -> Maybe a
```

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ExplicitForAll #-}

import Data.Kind

data Maybe :: Type -> Type where
 Nothing  :: forall (a :: Type). Maybe a
 Just     :: forall (a :: Type). a -> Maybe a
```

Kind	Description
Type	Proper types
Type -> Type	Unary type constructors

```
data Maybe a = Nothing | Just a

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

```
> :k Either
Either :: Type -> Type -> Type
```

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

> :k Either
Either :: Type -> Type -> Type

> :k Either String
Either String :: Type -> Type
```

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

> :k Either
Either :: Type -> Type -> Type

> :k Either String
Either String :: Type -> Type

> :k Either String Int
Either String Int :: Type
```

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

> :k Either
Either :: Type -> Type -> Type

> :k Either String
Either String :: Type -> Type

> :k Either String Int
Either String Int :: Type

> let e :: Either String Int = Right 3
```

Type -> Type -> Type

The kind of *binary* type constructors.
Types of this kind have no values.

e.g.
Either :: Type -> Type -> Type

Kind	Description
Type	Proper types
Type -> Type	Unary type constructors
Type -> Type -> Type	Binary type constructors (curried)

Java

Go

Kotlin

C#

Haskell

Scala

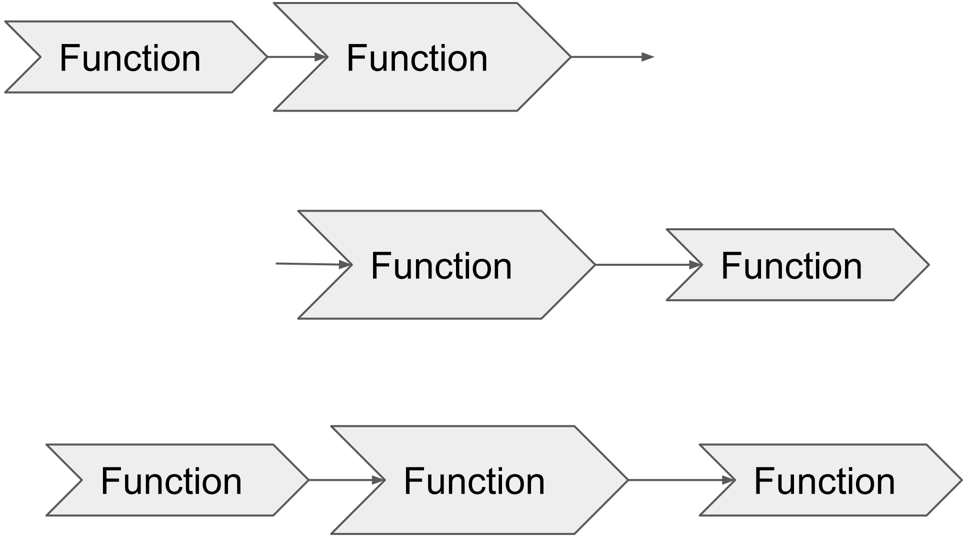
Idris

Agda

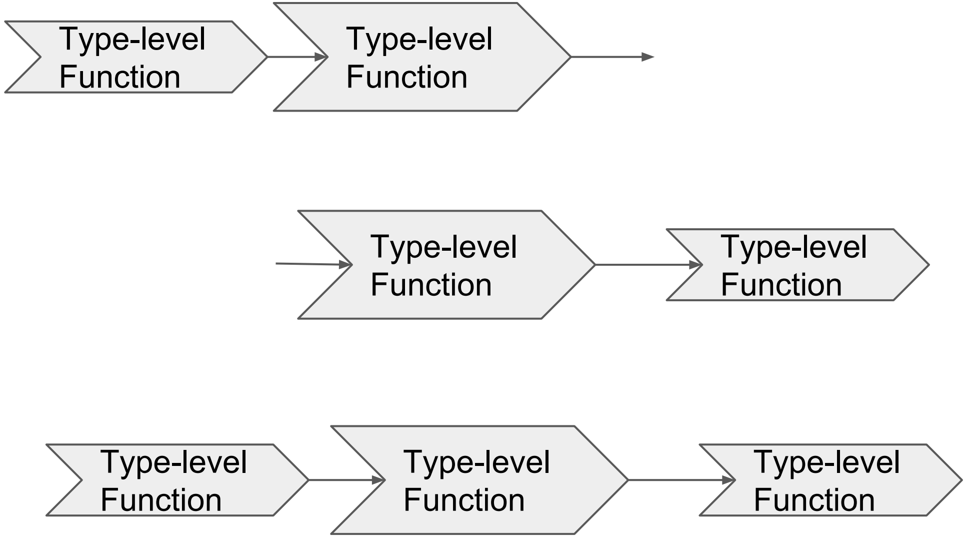
Coq

Higher Kinds

Higher-order function



Higher-kind



Type-level Function

Type-level Function

Type-level Function

Type-level Function

Type-level Function

Type-level Function

Type-level Function

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

```
> :kind Functor
Functor :: (Type -> Type) -> Constraint
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k Maybe
Maybe :: Type -> Type

> :k Functor Maybe
Functor Maybe :: Constraint
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k []
[] :: Type -> Type

> :k Functor []
Functor [] :: Constraint
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k IO
IO :: Type -> Type

> :k Functor IO
Functor IO :: Constraint
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k Either
Either :: Type -> Type -> Type

> :k Functor Either
<interactive>:1:9: error:
        • Expecting one more argument to ‘Either’
        Expected kind ‘Type -> Type’,
        but ‘Either’ has kind ‘Type -> Type -> Type’
        • In the first argument of ‘Functor’, namely ‘Either’
        In the type ‘Functor Either’
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k Either String
Either String :: Type -> Type

> :k Functor (Either String)
Functor (Either String) :: Constraint
```

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

> :k Functor
Functor :: (Type -> Type) -> Constraint

> :k forall a. Either a
forall a. Either a :: Type -> Type

> :k Functor (forall a. Either a)
Functor (forall a. Either a) :: Constraint
```

(Type -> Type) -> Constraint

The kind of single-parameter type classes in Haskell.
A higher kind.

e.g.
Functor :: (Type -> Type) -> Constraint

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE KindSignatures #-}

import Data.Kind

class Functor (f :: Type -> Type) where
 fmap :: (a -> b) -> f a -> f b
```

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ExplicitForAll #-}

import Data.Kind

class Functor (f :: Type -> Type) where
 fmap :: forall (a :: Type) (b :: Type). (a -> b) -> f a -> f b
```

```
trait Functor[F[ ]] {
 def fmap[A, B](fn: A => B, fa: F[A]): F[B]
}

scala> :k Functor
Functor's kind is X[F[A]]
```

(Type -> Type) -> Type

```
> :k Functor
Functor :: (Type -> Type) -> Constraint
```

```
trait Functor[F[ ]] {
 def fmap[A, B](fn: A => B, fa: F[A]): F[B]
}
```

```
class Functor (f :: Type -> Type) where
 fmap :: forall (a :: Type) (b :: Type). (a -> b) -> f a -> f b
```

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
trait Functor[F[ ]] {
 def fmap[A, B](fn: A => B, fa: F[A]): F[B]
}
```

kinds explicitly specified

```
class Functor (f :: Type -> Type) where
 fmap :: forall (a :: Type) (b :: Type). (a -> b) -> f a -> f b
```

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

kinds inferred

Type -> Type

```
trait Functor[F[ ]] {
 def fmap[A, B](fn: A => B, fa: F[A]): F[B]
}
```

Type

kinds explicitly specified

```
class Functor (f :: Type -> Type) where
 fmap :: forall (a :: Type) (b :: Type). (a -> b) -> f a -> f b
```

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

kinds inferred

Kind	Description
Type	Proper types
Type -> Type	Unary type constructors
Type -> Type -> Type	Binary type constructors (curried)
(Type -> Type) -> Constraint	Haskell type constructors
(Type -> Type) -> Type	Scala type constructor pattern

"Type" vs "*"

```
> :set -XStarIsType

> :k String
String :: *
```

Current defaults (GHC 8.6.1)

```
> :set -XNoStarIsType

> :k String
String :: Type
```

Future defaults.
Used for this talk.

Beyond...

Sorts (BOX)

Set, Set1, Set2, Set3...

Universes

TypeInType, Constraint Kinds, Poly Kinds, Data Kinds, Type Families

Dependent Types - Coq, Agda, Idris

Dependent Haskell

See also...

https://jozefg.bitbucket.io/posts/2014-02-10-types-kinds-and-sorts.html

https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/constraint-kind.html