Tour of the Typelevel

Jack Kelly
http://jackkelly.name/talks

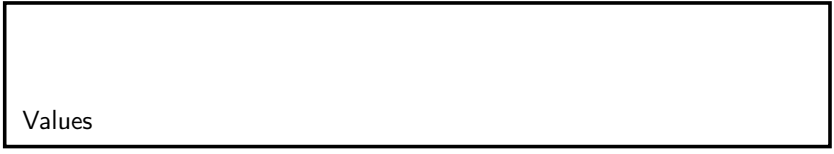
November 12, 2019

Tour of the Typelevel

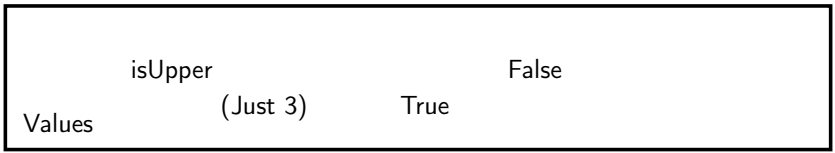
Tonight:

- Several common type system extensions.
- Motivating examples.
- Dropping signposts.
- Less scary when you see them in libraries.
- Please keep arms and legs inside the vehicle at all times.

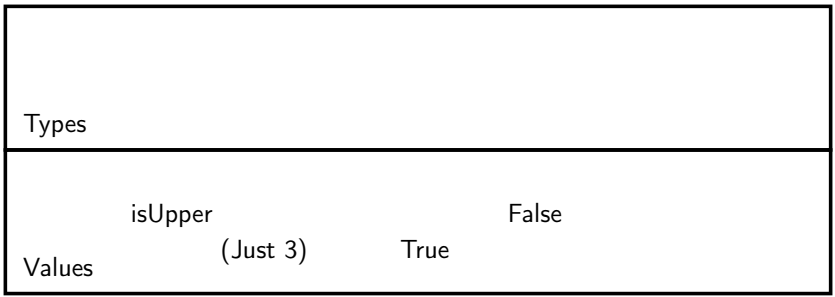
Values, Types, and Kinds



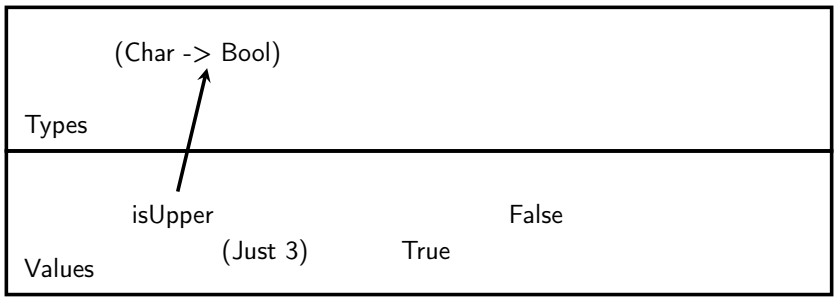
Values, Types, and Kinds



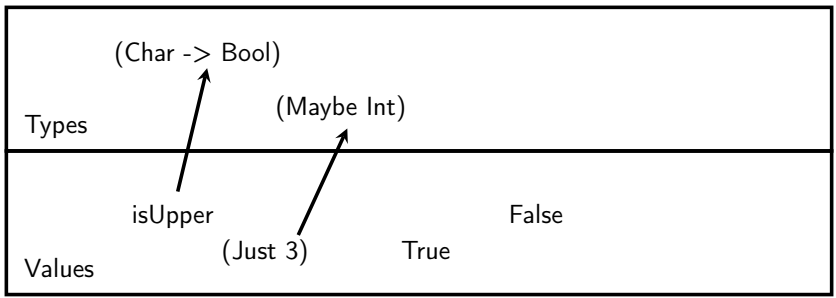
Values, Types, and Kinds



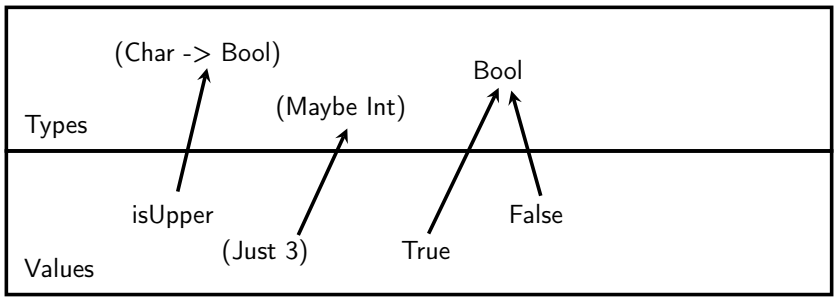
Values, Types, and Kinds



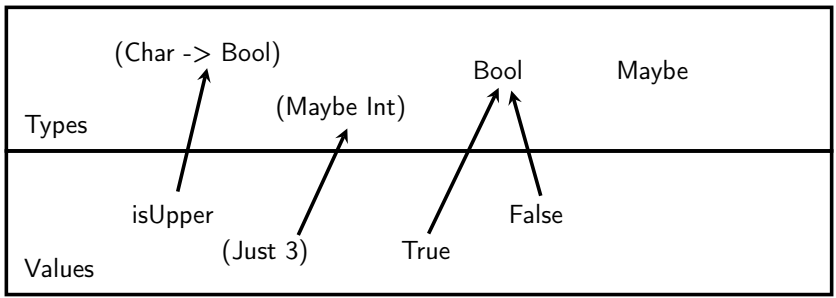
Values, Types, and Kinds



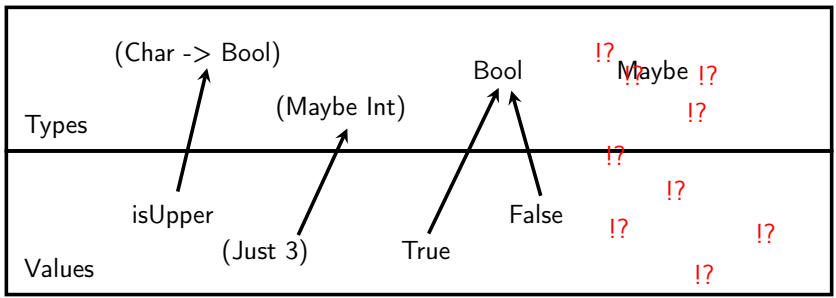
Values, Types, and Kinds



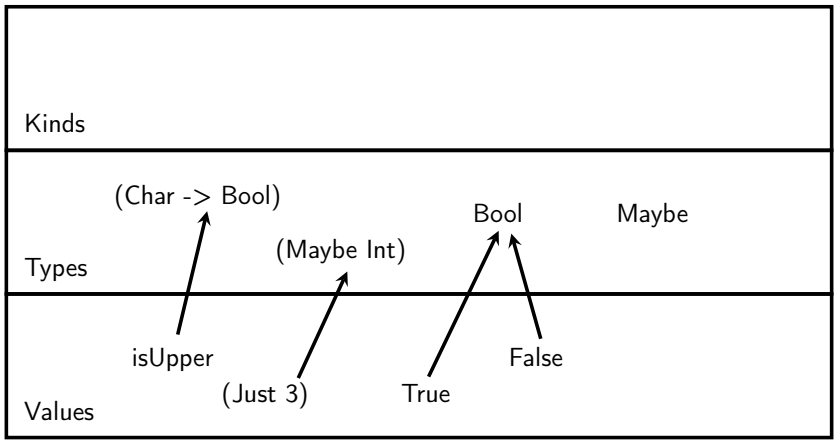
Values, Types, and Kinds



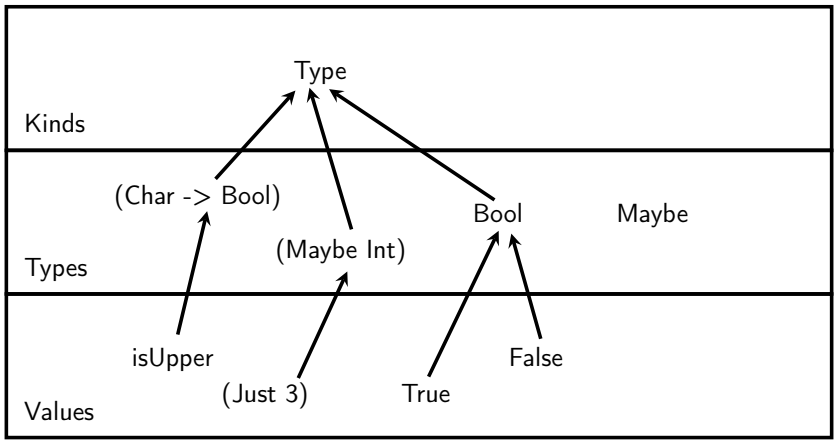
Values, Types, and Kinds



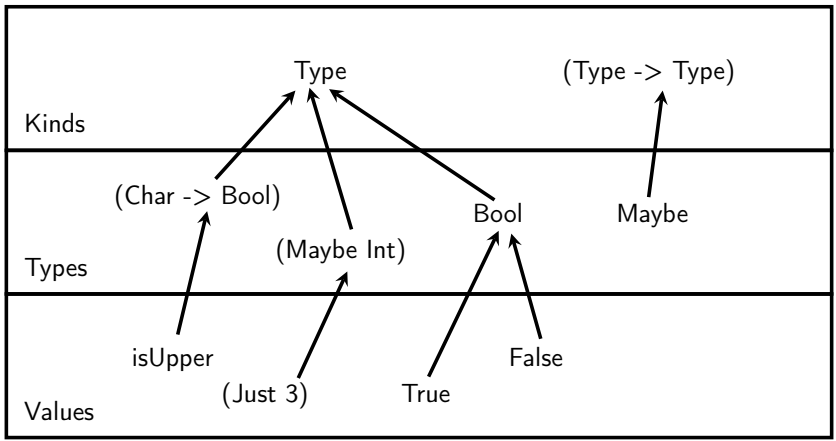
Values, Types, and Kinds



Values, Types, and Kinds



Values, Types, and Kinds



Values, Types, and Kinds

What did we learn?

- *Types* classify *values*.
- *Kinds* classify *types*.
- Types of kind `Data.Kind.Type` may have values.
  - Also called `*`, but this will (eventually) go away.
  - *May*, not *must*: `Void` has no values.
- Ask GHCi for the kind of a type:

```
> :kind Maybe
Maybe :: * -> *
```

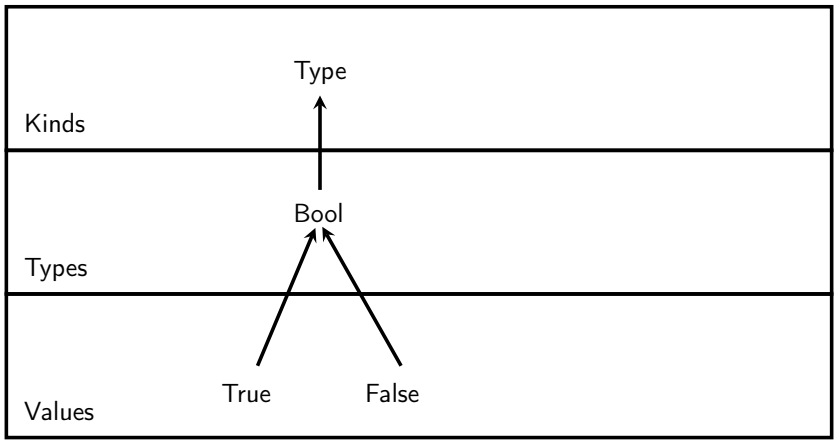
Kind Signatures

- Values may have *type signatures*.
- Types may have *kind signatures*:

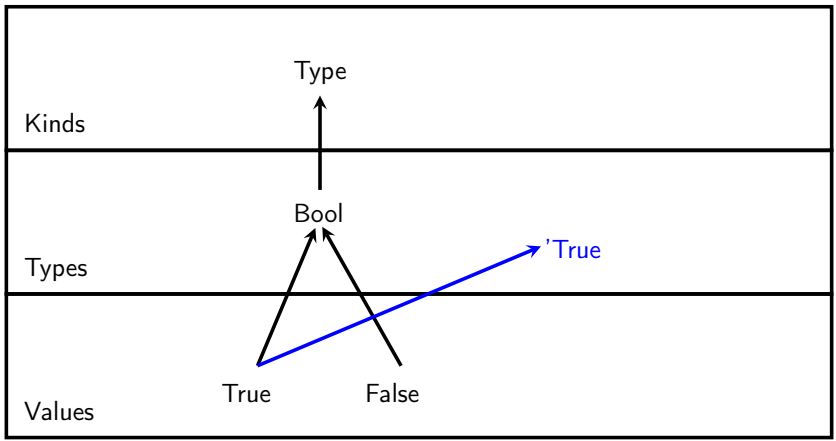
```
{-# LANGAUGE GADTSyntax, KindSignatures #-}

data Either (a :: Type) (b :: Type) :: Type where
  Left :: a -> Either a b
  Right :: b -> Either a b
```

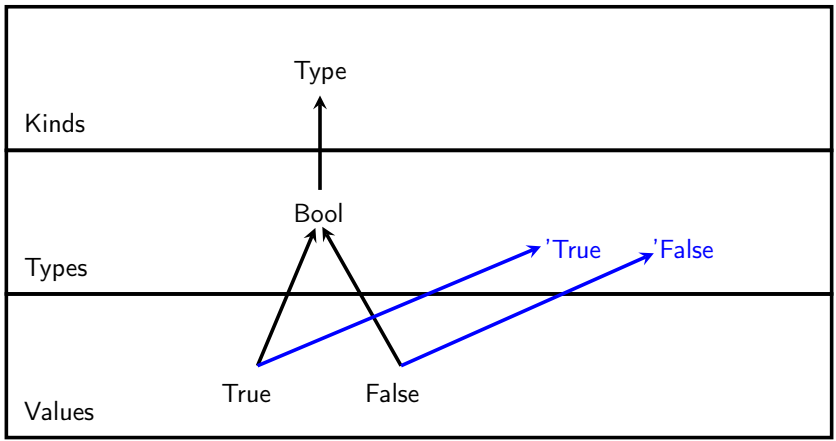
DataKinds



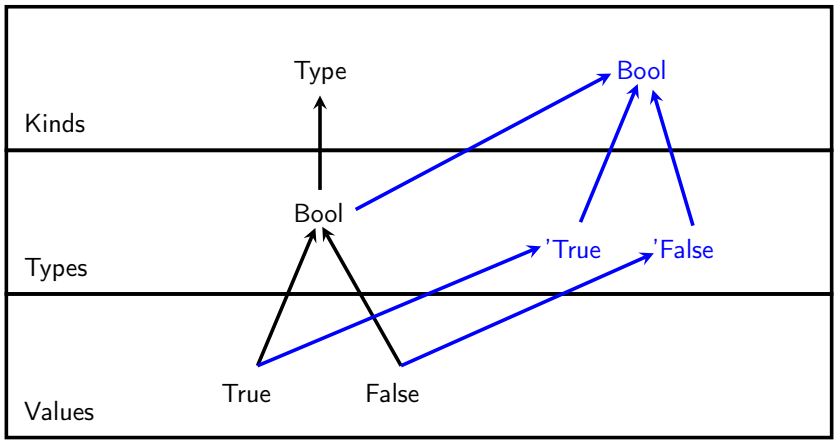
DataKinds



DataKinds



DataKinds



DataKinds

- What does `data Either a b = Left a | Right b` do?
- It introduces:
  - The type constructor `Either :: Type -> Type -> Type`
  - Two data constructors:
    - `Left :: a -> Either a b`
    - `Right :: b -> Either a b`

DataKinds

- What does `data Either a b = Left a | Right b` do?
- It introduces:
  - The type constructor `Either :: Type -> Type -> Type`
  - Two data constructors:
    - `Left :: a -> Either a b`
    - `Right :: b -> Either a b`
- If we set `{-# LANGUAGE DataKinds #-}`, we also get:
  - The *kind* `Either a b`
  - Two *type constructors*:
    - `'Left :: a -> Either a b`
    - `'Right :: b -> Either a b`

DataKinds

- What does `data Either a b = Left a | Right b` do?
- It introduces:
  - The type constructor `Either :: Type -> Type -> Type`
  - Two data constructors:
    - `Left :: a -> Either a b`
    - `Right :: b -> Either a b`
- If we set `{-# LANGUAGE DataKinds #-}`, we also get:
  - The *kind* `Either a b`
  - Two *type constructors*:
    - `'Left :: a -> Either a b`
    - `'Right :: b -> Either a b`
- `DataKinds` also gives you type-level strings (kind `Symbol`) and naturals (kind `Nat`).

DataKinds Example

Suppose:

- I have a data structure, stored on disk.
- It is *really important* that I verify it before use.
  - e.g. Moxie's Cryptographic Doom Principle
    `https://moxie.org/blog/the-cryptographic-doom-principle/`
- How do I enforce this?

DataKinds Example: Unsafe API

```
data Structure = -- ...

loadStructure :: FilePath -> IO Structure
loadStructure = -- ...

verify :: Structure -> Bool
verify = -- ...
```

DataKinds Example: Verified Reads

```haskell
{-# LANGUAGE DataKinds #-}

data Verification = Unverified | Verified
data Structure (v :: Verification) = -- ...
```

DataKinds Example: Verified Reads

```haskell
{-# LANGUAGE DataKinds #-}

data Verification = Unverified | Verified
data Structure (v :: Verification) = -- ...

loadStructure
  :: FilePath
  -> IO (Structure 'Unverified)
loadStructure = -- ...
```

DataKinds Example: Verified Reads

```haskell
{-# LANGUAGE DataKinds #-}

data Verification = Unverified | Verified
data Structure (v :: Verification) = -- ...

loadStructure
  :: FilePath
  -> IO (Structure 'Unverified)
loadStructure = -- ...

verify
  :: Structure v
  -> Maybe (Structure 'Verified)
verify = -- ...
```

DataKinds Example: HList

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

data HList (xs :: [Type]) :: Type where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

DataKinds Example: HList

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

data HList (xs :: [Type]) :: Type where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

DataKinds Example: HList

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

data HList (xs :: [Type]) :: Type where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

DataKinds Example: HList

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

data HList (xs :: [Type]) :: Type where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

DataKinds Example: HList

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

data HList (xs :: [Type]) :: Type where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

```
> :t HCons True (HCons "Hello" HNil)
HCons True (HCons "Hello" HNil) :: HList '[Bool, [Char]]
```

ConstraintKinds

`{-# LANGUAGE ConstraintKinds #-}` lets us use the kind `Constraint` in contexts:

- It gives us a new kind called `Constraint`:

```
> :k Show
Show :: * -> Constraint
> :k Monad
Monad :: (* -> *) -> Constraint
```

ConstraintKinds

`{-# LANGUAGE ConstraintKinds #-}` lets us use the kind `Constraint` in contexts:

- It gives us a new kind called `Constraint`:

```
> :k Show
Show :: * -> Constraint
> :k Monad
Monad :: (* -> *) -> Constraint
```

- Tuples of `Constraint` are also of kind `Constraint`:

```
> :k (Eq Int, Show Int)
(Eq Int, Show Int) :: Constraint
```

ConstraintKinds

- We can write constraint aliases using `type`:

```
type Stringy a = (Read a, Show a)
roundtrip :: Stringy a => a -> a
roundtrip = read . show
```

- Ed Kmett has a library `constraints` for playing with these, and a talk "Type Classes vs. the World" (skip to 46:00).

Multi-Parameter Type Classes

- What is a type class?
  - It's a property on types:
    - `Int` is in `Num`, `Show`, ...
  - It may have methods for ad-hoc polymorphism:
    - Implementation chosen by the argument type
- Why restrict ourselves to one type?

Multi-Parameter Type Classes

- Set `{-# LANGUAGE MultiParamTypeClasses #-}`.
- Type classes are now *N*-ary relations on types:

```
class Resource idType resourceType where
  url :: idType -> String
  get :: idType -> IO resourceType

newtype PostId = -- ...
data Post = -- ...
instance Resource PostId Post where
  -- ...
```

Multi-Parameter Type Classes

- Set `{-# LANGUAGE MultiParamTypeClasses #-}`.
- Type classes are now N-ary relations on types:

```
class Resource idType resourceType where
  url :: idType -> String
  get :: idType -> IO resourceType

newtype PostId = -- ...
data Post = -- ...
instance Resource PostId Post where
  -- ...
```

- What is the type of `get (pid :: PostId)`?
  - We want `IO Post`.
  - But type classes are an open world!
  - How can GHC know there's no other instances?

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

- What is the type of `get (pid :: PostId)`?

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

- What is the type of `get (pid :: PostId)`?
  - `IO Post`.

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

- What is the type of `get (pid :: PostId)`?
  - `IO Post`.
- What is the type of `pid` in `get pid :: IO Post`?

Functional Dependencies

- Set `{-# LANGUAGE FunctionalDependencies #-}`.
- We can now say which parameters force which other parameters:

```
class Resource i r | i -> r, r -> i where
  url :: i -> String
  get :: i -> IO r

newtype PostId = -- ...
data Post = -- ...
instance PostId Post where
  -- ...
```

- What is the type of `get (pid :: PostId)`?
  - `IO Post`.
- What is the type of `pid` in `get pid :: IO Post`?
  - `PostId`.

Type Families

- Type families are type-level functions.
- Think of them as type aliases that can inspect their argument.
- Set `{-# LANGUAGE TypeFamilies #-}`

Type Families: Closed Type Families

- The simplest:

```
type family
  Append (xs :: [k]) (ys :: [k]) :: [k]
  where

  Append '[] ys = ys
  Append (x ': xs) ys = x ': (Append xs ys)
```

Type Families: Closed Type Families

- The simplest:

```
type family
  Append (xs :: [k]) (ys :: [k]) :: [k]
  where

  Append '[] ys = ys
  Append (x ': xs) ys = x ': (Append xs ys)
```

- `:kind!` will force GHCi to expand a type-level expression:

```
> :kind! Append '[Int] '[Bool]
Append '[Int] '[Bool] :: [*]
= '[Int, Bool]
```

Type Families: Open Type Families

- Instead of providing all the equations, allow anyone to add their own.

```
{-# LANGUAGE DataKinds     #-}
{-# LANGUAGE PolyKinds     #-}
{-# LANGUAGE TypeFamilies #-}

data Permission = P1 | P2
type family
  Permissions (act :: k) :: [Permission]

-- In some other module...
data Action = A1 | A2
type instance Permissions 'A1 = ['P1, 'P2]
type instance Permissions 'A2 = '[]
```

Type Families: Associated Types

- Add type aliases to type classes.

```
class MonoFunctor m where
  type Elem m :: Type
  omap :: (Elem m -> Elem m) -> m -> m

instance MonoFunctor Text where
  type Elem Text = Char
  omap = -- ...
```

- Remark: This is a good motivating example for associated types, but `Traversals` (from `lens`) are a better solution to this problem.

Data Families

- Also turned on by `{-# LANGUAGE TypeFamilies #-}`.
- Write `data` instead of `type`: `data family ... where` etc.
- Let you create new data types based on type arguments.
- Used much less often.

Proxy

- Consider this function:

```
normalise :: String -> String
normalise = show . read
```

- What instances of `Read` and `Show` do we use?

Proxy

- Consider this function:

```
normalise :: String -> String
normalise = show . read
```

- What instances of `Read` and `Show` do we use?
- We can use a `Proxy` to explicitly choose an instance.

```
{-# LANGUAGE PolyKinds, KindSignatures #-}

data Proxy (a :: k) = Proxy
```

Proxy Example: String Normalisation

- Consider this version:

```
{-# LANGUAGE ScopedTypeVariables #-}

normalise
  :: forall a . (Read a, Show a)
  => Proxy a
  -> String
  -> String
normalise _ s = show (read s :: a)
```

Proxy Example: Value of a Symbol at Runtime

```haskell
{-# LANGUAGE DataKinds, TypeApplications #-}

import Data.Proxy
import GHC.TypeLits

hello :: String
hello = symbolVal (Proxy :: Proxy "hello")

hello' :: String
hello' = symbolVal (Proxy @"hello")
```

Summary

- `DataKinds` lifts `data` declarations into new types and kinds.
- `ConstraintKinds` lets you use types of kind `Constraint` inside contexts.
- `GADTs` let you fiddle the type variables in data constructors.
- `MultiParamTypeClasses` turn type classes into *relations* on types. Use `FunctionalDependencies` to help things along.
- `ScopedTypeVariables` lets you use type variables inside function bodies.
- `TypeFamilies` lets you write type-level functions.
- The `Proxy` type lets you pass additional type information to functions.