GHC LANGUAGE EXTENSIONS

Andrew McMiddlin

2019-05-15

```
type-class-extensions.lhs:3:3: error:
    • Too many parameters for class ‘Foo’
      (Enable MultiParamTypeClasses to allow multi-parameter classes)
    • In the class declaration for ‘Foo’
  |
3 | > class Foo a b where
  |   ^^^^^^^^^^^^^^^^^^...
```

```
type-class-extensions.lhs:3:3: error:
    • Too many parameters for class ‘Foo’
      (Enable MultiParamTypeClasses to allow multi-parameter classes)
    • In the class declaration for ‘Foo’
  |
3 | > class Foo a b where
  |   ^^^^^^^^^^^^^^^^^^...
```


{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE CPP #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DerivingVia #-}
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE EmptyCase #-}
{-# LANGUAGE EmptyDataDecls #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE GeneralisedNewtypeDeriving #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE OverloadedLabels #-}
{-# LANGUAGE OverloadedLists #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE QuantifiedConstraints #-}
{-# LANGUAGE Rank2Types #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE RebindableSyntax #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE RecursiveDo #-}
{-# LANGUAGE RoleAnnotations #-}
{-# LANGUAGE Safe #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeInType #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE UnboxedSums #-}
LANGUAGE EXTENSIONS 101

HASKELL 2010

Haskell 2010 is defined in the Haskell 2010 Language Report.

WHAT’S NOT IN HASKELL 2010?

WHAT'S NOT IN HASKELL 2010?

- Type classes with more than one parameter.

WHAT'S NOT IN HASKELL 2010?

- Type classes with more than one parameter.
- String literals for anything other than `[Char]`

WHAT'S NOT IN HASKELL 2010?

- Type classes with more than one parameter.
- String literals for anything other than `[Char]`
- Generalised Algebraic Data Types (GADTs)

LANGUAGE EXTENSIONS

LANGUAGE EXTENSIONS

Section 12.3 covers the LANGUAGE pragma, which is used for extensions.

ENABLING EXTENSIONS IN GHC

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE GADTs, ScopedTypeVariables #-}
```

```
default-extensions:    OverloadedStrings
                     , GADTs
                     , ScopedTypeVariables
```

```
ghc -XOverloadedStrings Foo.hs
```

```
$ ghci
λ :set -XOverloadedStrings
```


SUGAR

`OverloadedStrings`

Enable overloaded string literals.

```
GHCi, version 8.6.4: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/andrew/git/dot-files/.ghci
λ> :t "Lambda"
"Lambda" :: [Char]
```

```
GHCi, version 8.6.4: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/andrew/git/dot-files/.ghci
λ> :t "Lambda"
"Lambda" :: [Char]
λ> :set -XOverloadedStrings
λ> :t "Jam"
"Jam" :: Data.String.IsString p => p
```

```haskell
class IsString a where
  fromString :: String -> a
```

```haskell
class IsString a where
  fromString :: String -> a

instance IsString Text where
  fromString = pack
```

```haskell
class IsString a where
  fromString :: String -> a

instance IsString Text where
  fromString = pack

isGood :: Text -> Bool
```

```haskell
class IsString a where
  fromString :: String -> a

instance IsString Text where
  fromString = pack

isGood :: Text -> Bool

isGood "foo"
```

`TupleSections`

Allow partially applied tuple constructors.

```
\x -> x * 2
```

```haskell
\x -> x * 2

(* 2)
```

```haskell
\x -> x * 2

(* 2)

\x -> (x,True)
```

```haskell
\x -> x * 2

(* 2)

\x -> (x,True)

(,True)
```

```
(,True,,,"hi",)
```

```
(,True,,,"hi",) :: a -> b -> c -> d -> (a,Bool,b,c,String,d)
```

`InstanceSigs`

Allow type signatures for definitions of instance members.

```
instance (Traversable f, Traversable g) => Traversable (Compose f g)

  traverse = undefined
```

```
instance (Traversable f, Traversable g) => Traversable (Compose f g)
  traverse :: (a -> h b) -> Compose f g a -> h (Compose f g b)
  traverse = undefined
```

```
    • Illegal type signature in instance declaration:
        traverse' :: (a -> h b) -> Compose f g a -> h (Compose f g b)
      (Use InstanceSigs to allow this)
    • In the instance declaration for ‘Traversable' (Compose f g)’
   |
25 |   traverse' :: (a -> h b) -> Compose f g a -> h (Compose f g b)
   |                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

`LambdaCase`

Adds syntactic sugar for pattern matching on a function's argument.

```haskell
pretty ::
  -> Expr
  -> Text
pretty e = case e of
  LitI n -> pack $ show n
  LitB True -> "true"
  LitB False -> "false"
```

```
pretty ::
  -> Expr
  -> Text
pretty = \case
  LitI n -> pack $ show n
  LitB True -> "true"
  LitB False -> "false"
```

MultiWayIf

Adds syntactic sugar for nested `if-then-else` expressions.

```
if 1 < 0 then
  "foo"
else if 12 > 4 then
  "bar"
else if even 42 then
  "42"
else
  "no idea"
```

```
if | 1 < 0 -> "foo"
   | 12 > 4 -> "bar"
   | even 42 -> "42"
   | otherwise -> "no idea"
```


RECORDS

`RecordWildCards`

Elide fields from record construction and pattern matching.

```
data Person =
  Person {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  }
```

```haskell
data Person =
  Person {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  }

greetPerson ::
  Person
  -> Text
greetPerson Person{firstName = firstName, surname = surname, height = height} =
  undefined
```

```haskell
data Person =
  Person {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  }

greetPerson ::
  Person
  -> Text
greetPerson Person{firstName = firstName, surname = surname, height = height} =
  undefined
```

```haskell
{-# LANGUAGE RecordWildCards #-}

data Person =
  Person {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  }

greetPerson ::
  Person
  -> Text
greetPerson Person{firstName = firstName, surname = surname, height = height} =
  undefined
```

```haskell
{-# LANGUAGE RecordWildCards #-}

data Person =
  Person {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  }

greetPerson ::
  Person
  -> Text
greetPerson Person{..} =
  undefined
```

```haskell
{-# LANGUAGE RecordWildCards   #-}

defaultPerson ::
  Person
defaultPerson =
  let
    firstName = "Andrew"
    surname = "McMiddlin"
    height = 185
  in
    Person {..}
```

```
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE RecordWildCards   #-}
```

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE RecordWildCards   #-}

data ConferenceAttendee =
  ConferenceAttendee {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  , shirtSize :: ShirtSize
  }
```

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE RecordWildCards   #-}

data ConferenceAttendee =
  ConferenceAttendee {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  , shirtSize :: ShirtSize
  }

defaultConferenceAttendee ::
  Person
  -> ConferenceAttendee
defaultConferenceAttendee           =
```

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE RecordWildCards   #-}

data ConferenceAttendee =
  ConferenceAttendee {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  , shirtSize :: ShirtSize
  }

defaultConferenceAttendee ::
  Person
  -> ConferenceAttendee
defaultConferenceAttendee Person{..} =
```

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE RecordWildCards   #-}

data ConferenceAttendee =
  ConferenceAttendee {
    firstName :: Text
  , surname   :: Text
  , height    :: Integer
  , shirtSize :: ShirtSize
  }

defaultConferenceAttendee ::
  Person
  -> ConferenceAttendee
defaultConferenceAttendee Person{..} =
  ConferenceAttendee {shirtSize = M, ..}
```

Some problems with `RecordWildCards`

Some problems with `RecordWildCards`

- Unclear where variables come from.

Some problems with `RecordWildCards`

- Unclear where variables come from.
- All fields are brought into scope.

Some problems with `RecordWildCards`

- Unclear where variables come from.
- All fields are brought into scope.
- Vulnerable to changes in the record.

NamedFieldPuns

Remove some of the boilerplate when bringing record fields into scope.

```haskell
{-# LANGUAGE NamedFieldPuns #-}

greetPerson ::
  Person
  -> Text
greetPerson                                =
  undefined
```

```haskell
{-# LANGUAGE NamedFieldPuns #-}

greetPerson ::
  Person
  -> Text
greetPerson Person{firstName, surname, height} =
  undefined
```

```haskell
{-# LANGUAGE NamedFieldPuns #-}

greetPerson ::
  Person
  -> Text
greetPerson Person{firstName, surname} =
  undefined
```


HEAVYWEIGHT

ScopedTypeVariables

Scope type variables to the lexical scope of the expression.

```haskell
f ::
  [a] -> [a]
f xs =
  ys ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

```haskell
f ::
  [a] -> [a]
f xs =
  ys ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

```
Couldn't match type ‘a’ with ‘a1’
‘a’ is a rigid type variable bound by
  the type signature for:
    f :: forall a. [a] -> [a]
  at examples/ScopedTypeVariables.hs:(5,1)-(6,12)
‘a1’ is a rigid type variable bound by
  the type signature for:
    ys :: forall a1. [a1]
  at examples/ScopedTypeVariables.hs:10:5-13
Expected type: [a1]
  Actual type: [a]
```

```
Couldn't match type ‘a’ with ‘a1’
‘a’ is a rigid type variable bound by
  the type signature for:
    f :: forall a. [a] -> [a]
  at examples/ScopedTypeVariables.hs:(5,1)-(6,12)
‘a1’ is a rigid type variable bound by
  the type signature for:
    ys :: forall a1. [a1]
  at examples/ScopedTypeVariables.hs:10:5-13
Expected type: [a1]
  Actual type: [a]
```

```haskell
f ::

  [a] -> [a]
f xs =
  ys ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

f ::

  [a] -> [a]
f xs =
  ys ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

f ::
  forall a.
  [a] -> [a]
f xs =
  ys ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

`GeneralisedNewtypeDeriving`

Derive instances for `newtype`s based on the type they wrap.

```haskell
class Pretty a where
  pretty :: a -> Text
```

```haskell
class Pretty a where
  pretty :: a -> Text

instance Pretty Int where
  pretty = pack . show
```

```haskell
class Pretty a where
  pretty :: a -> Text

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
```

```haskell
class Pretty a where
  pretty :: a -> Text

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)
```

```
Can't make a derived instance of ‘Pretty Age’:
  ‘Pretty’ is not a stock derivable class (Eq, Show, etc.)
  Try GeneralizedNewtypeDeriving for GHC's newtype-deriving extension
```

```
class Coercible a b
```

```haskell
class Coercible a b

coerce :: Coercible a b => a -> b
```

```haskell
instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)
```

```haskell
{-# LANGUAGE GeneralisedNewtypeDeriving #-}

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)
```

```haskell
{-# LANGUAGE GeneralisedNewtypeDeriving #-}

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)

instance Coercible Int Age
instance Coercible Age Int
```

```haskell
{-# LANGUAGE GeneralisedNewtypeDeriving #-}

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)

instance Coercible Int Age
instance Coercible Age Int

instance Pretty Age where
  pretty = coerce $ pack . show
```

```haskell
{-# LANGUAGE GeneralisedNewtypeDeriving #-}

instance Pretty Int where
  pretty = pack . show

newtype Age = Age Int
  deriving (Show, Pretty)

instance Coercible Int Age
instance Coercible Age Int

instance Pretty Age where
  pretty = coerce $ pack . show

instance Coercible a b => Coercible (a -> c) (b -> c)
```

ROLES

ROLES

`GeneralisedNewtypeDeriving` as it was originally implemented had some issues that resulted in **roles** being added to the language.

ROLES

`GeneralisedNewtypeDeriving` as it was originally implemented had some issues that resulted in **roles** being added to the language.

As a result of the role system, adding `join` to the `Monad` class would stop `GeneralisedNewtypeDeriving` from being able to derive `Monad`.

TYPE CLASSES

TYPE CLASSES IN HASKELL 2010

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

- *must* have the `class` keyword;

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

- *must* have the `class` keyword;
- *may* have a context;

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

- *must* have the `class` keyword;
- *may* have a context;
- *must* have a class name;

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

- *must* have the `class` keyword;
- *may* have a context;
- *must* have a class name;
- *must* be parameterised over exactly one type; and

TYPE CLASSES IN HASKELL 2010

Section 4.3.1 of the standard covers type classes.

To summarise, it says that a type class declaration must have the following form.

```
class cx => C u where cdecls
```

- *must* have the `class` keyword;
- *may* have a context;
- *must* have a class name;
- *must* be parameterised over exactly one type; and
- *may* declare one or more members.

```haskell
class Show a where
  show :: a -> String
  ...
```

```haskell
class Show a where
  show :: a -> String
  ...

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  ...
```

```haskell
class Show a where
  show :: a -> String
  ...

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  ...

class (Ord a, Show a) => ShOrd a
```

TYPE CLASS INSTANCES IN HASKELL 2010

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

- *must* start with the `instance` keyword;

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

- *must* start with the `instance` keyword;
- *may* have a context;

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

- *must* start with the `instance` keyword;
- *may* have a context;
- *must* mention the class name;

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

- *must* start with the `instance` keyword;
- *may* have a context;
- *must* mention the class name;
- *must* mention the type the instance is for; and

TYPE CLASS INSTANCES IN HASKELL 2010

Section 4.3.2 of the standard covers type class instance declarations.

In short, it says that a type class instance must have the following form.

```
instance cx => C (T u1 … uk) where { d }
```

- *must* start with the `instance` keyword;
- *may* have a context;
- *must* mention the class name;
- *must* mention the type the instance is for; and
- *may* contain definitions for the class's members.

MultiParamTypeClasses

Allows type classes with more than one type parameter.

```haskell
class Monad m => MonadReader r m where
  ask :: m r
  ...
```

`FlexibleInstances`

Relaxes the rules for valid type class instances.

- Instance types can be type variables.

- Instance types can be type variables.
- Type variables can appear multiple times in the instance head.

- Instance types can be type variables.
- Type variables can appear multiple times in the instance head.
- Concrete types may be used as parameters to instance types.

```haskell
class Monad m => MonadReader r m where
  ask :: m r
```

```haskell
class Monad m => MonadReader r m where
  ask :: m r

instance MonadReader r ((->) r) where
  ask = id
```

```
type-class-extensions.lhs:123:10-32: error:
    • Illegal instance declaration for ‘MonadReader r ((->) r)’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance head.
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘MonadReader r ((->) r)’
    |
123 | instance MonadReader r ((->) r) where
```

```
type-class-extensions.lhs:123:10-32: error:
    • Illegal instance declaration for ‘MonadReader r ((->) r)’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance head.
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘MonadReader r ((->) r)’
    |
123 | instance MonadReader r ((->) r) where
```

```
type-class-extensions.lhs:123:10-32: error:
    • Illegal instance declaration for ‘MonadReader r ((->) r)’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance head.
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘MonadReader r ((->) r)’
    |
123 | instance MonadReader r ((->) r) where
```

```
type-class-extensions.lhs:123:10-32: error:
    • Illegal instance declaration for ‘MonadReader r ((->) r)’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance head.
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘MonadReader r ((->) r)’
    |
123 | instance MonadReader r ((->) r) where
```

```haskell
class Twizzle a where
  twizzle :: a -> Int

instance Twizzle (Maybe Integer) where
  twizzle = maybe 42 fromInteger
```

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.4
```

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.4

$ ghc -Wall -fforce-recomp Main.hs -o whoopsie
```

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.4

$ ghc -Wall -fforce-recomp Main.hs -o whoopsie
[1 of 4] Compiling FIA              ( FIA.hs, FIA.o )
[2 of 4] Compiling FIB              ( FIB.hs, FIB.o )
[3 of 4] Compiling FIC              ( FIC.hs, FIC.o )
[4 of 4] Compiling Main             ( Main.hs, Main.o )
Linking whoopsie ...
```

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.4

$ ghc -Wall -fforce-recomp Main.hs -o whoopsie
[1 of 4] Compiling FIA              ( FIA.hs, FIA.o )
[2 of 4] Compiling FIB              ( FIB.hs, FIB.o )
[3 of 4] Compiling FIC              ( FIC.hs, FIC.o )
[4 of 4] Compiling Main             ( Main.hs, Main.o )
Linking whoopsie ...

> ./whoopsie
fromList [Whoopsie A1 B C,Whoopsie A2 B C,Whoopsie A1 B C]
```

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 8.4.4

$ ghc -Wall -fforce-recomp Main.hs -o whoopsie
[1 of 4] Compiling FIA              ( FIA.hs, FIA.o )
[2 of 4] Compiling FIB              ( FIB.hs, FIB.o )
[3 of 4] Compiling FIC              ( FIC.hs, FIC.o )
[4 of 4] Compiling Main             ( Main.hs, Main.o )
Linking whoopsie ...

> ./whoopsie
fromList [Whoopsie A1 B C,Whoopsie A2 B C,Whoopsie A1 B C]
```

`FlexibleContexts`

Relax some of the requirements regarding contexts.

```
updateThing ::
  MonadState MyState m
  => m ()
```

```haskell
updateThing ::
  ( HasThing s
  , MonadState s m
  )
  => m ()
```

FunctionalDependencies

Express dependent relationships between type variables for type classes with multiple parameters.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

class Monad m => MonadReader r m where
  ask :: m r
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

class Monad m => MonadReader r m where
  ask :: m r

instance MonadReader r ((->) r) where
  ask = id
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

class Monad m => MonadReader r m where
  ask :: m r

instance MonadReader r ((->) r) where
  ask = id

foo ::
  Integer
foo =
  (+ 1) <$> ask $ 41
```

```
type-class-extensions.lhs:275:13-16: error:
    • Ambiguous type variable ‘t0’ arising from a use of ‘ask’
      prevents the constraint ‘(MonadReader
                                  Integer ((->) t0))’ from being solved.
      Probable fix: use a type annotation to specify what ‘t0’ should be.
      These potential instance exist:
        one instance involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the second argument of ‘(<$>)’, namely ‘ask’
      In the expression: (+ 1) <$> ask
      In the expression: (+ 1) <$> ask $ 100
    |
275 |   (+ 1) <$> ask $ 41
    |              ^^^
```

```
type-class-extensions.lhs:275:13-16: error:
    • Ambiguous type variable ‘t0’ arising from a use of ‘ask’
      prevents the constraint ‘(MonadReader
                                  Integer ((->) t0))’ from being solved.
      Probable fix: use a type annotation to specify what ‘t0’ should be.
      These potential instance exist:
        one instance involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the second argument of ‘(<$>)’, namely ‘ask’
      In the expression: (+ 1) <$> ask
      In the expression: (+ 1) <$> ask $ 100
    |
275 |   (+ 1) <$> ask $ 41
    |              ^^^
```

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadReader r m | m -> r where
  ask :: m r
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadReader r m | m -> r where
  ask :: m r
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadReader r m | m -> r where
  ask :: m r

instance MonadReader r ((->) r) where
  ask = id
```

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadReader r m | m -> r where
  ask :: m r

instance MonadReader r ((->) r) where
  ask = id

foo ::
  Integer
foo =
  (+ 1) <$> ask $ 41
```

CONCLUSION

- Haskell 2010 is smaller than you think.

- Haskell 2010 is smaller than you think.
- GHC defines many extensions to the language.

- Haskell 2010 is smaller than you think.
- GHC defines many extensions to the language.
- Language extensions come with tradeoffs.

REFERENCES

GHC language extensions
https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html

Haskell 2010 report
https://www.haskell.org/onlinereport/haskell2010/haskellch12.html#x19-19100012.3

24 Days of GHC extensions
https://ocharles.org.uk/pages/2014-12-01-24-days-of-ghc-extensions.html

Putting `join` in `Monad`
https://ryanglscott.github.io/2018/03/04/how-quantifiedconstraints-can-let-us-put-join-back-in-monad/

`FlexibleInstances` breaking `Data.Set`
https://gist.github.com/rwbarton/dd8e51dce2a262d17a80

IMAGES

Muhammad Ali
https://commons.wikimedia.org/wiki/File:Muhammad_Ali_1966.jpg

Records
https://flic.kr/p/8fsrnG