Overlapping and Undecidable and Incoherent, Oh My!

by
Jake Keuhlen
github.com/jkeuhlen/talks

Overview

- Typeclass Crash Course
- “Class and Instance Declarations”

What is a typeclass?

“Haskell uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics, but the type system has been extended with type classes (or just classes) that provide a structured way to introduce overloaded functions”

Haskell2010 Report

What is a typeclass?

Type classes are a type indexed dictionary of function definitions.

Given a type, GHC will look up the implementation of a function based on the instances in scope.

Type classes are also a way to overload functions based on their type. Instances behave differently depending on the context they are used in.

What is a typeclass?

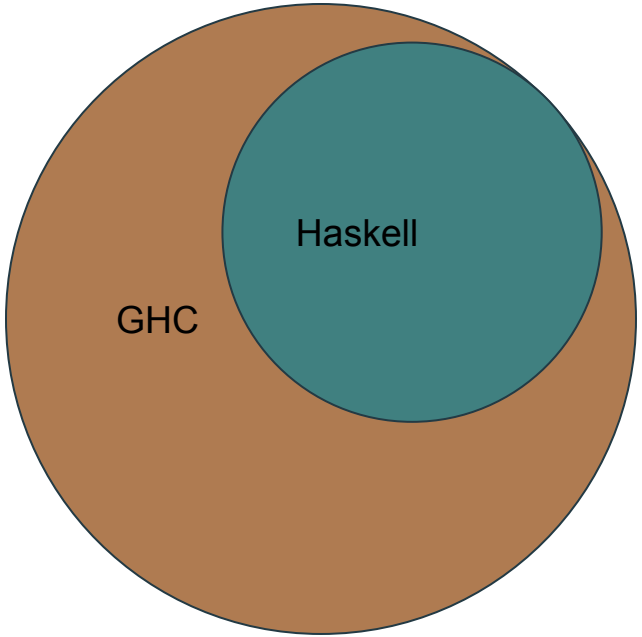
As defined in the Haskell report, type classes are *simple.*

“When type classes were first introduced in Haskell they were regarded as a fairly experimental language feature, and therefore warranted a fairly conservative design.”

“All programmers want more than they are given, and many people have bumped up against the limitations of Haskell's class system.”

- Peyton Jones, Jones, and Meijer ‘97

Detour: Haskell vs GHC



Class extensions

- MultiParamTypeClasses
- FlexibleContexts
- ConstrainedClassMethods
- DefaultSignatures
- NullaryTypeClasses
- FunctionalDependencies

Instance Extensions

- TypeSynonymInstances
- FlexibleInstances
- UndecidableInstances
- OverlappingInstances
- IncoherentInstances
- InstanceSigs
- OverloadedStrings
- OverloadedLabels
- OverloadedLists
- UndecidableSuperClasses

Class and Instance Extensions

1. What
2. Why
3. How
4. When

Class extensions

- MultiParamTypeClasses
- FlexibleContexts
- ConstrainedClassMethods
- DefaultSignatures
- NullaryTypeClasses
- FunctionalDependencies

Side Note

Everything in the coming sections that is written in ***bold italics*** is taken verbatim from the GHC User's Guide that describes these features.

Many of the code examples are also taken or derived from the same source.

Multi-parameter type classes - What

Allow the definition of typeclasses with more than one parameter.

Multi-parameter type classes - Why

Type classes started out as an experimental language feature, and were designed more conservatively than necessary to ensure various guarantees about the language. As is true with most language extensions present in GHC, using only the features present in Haskell98 or Haskell2010 could end up feeling overly restrictive, and programmers want more flexibility in natural applications of tools. Multiple parameter type classes are just one such use case.

Multi-parameter type classes - How

```
class MPTCE f a where
  update :: (a -> b) -> f a -> f b

-- src\MultiParamTypeClasses.hs:3:1: error:
--     * Too many parameters for class `MPTCE'
--       (Enable MultiParamTypeClasses to allow multi-parameter classes)
--     * In the class declaration for `MPTCE'
--   |
-- 3 | class MPTCE f a where
--   | ^^^^^^^^^^^^^^^^^^^^^...

instance Functor f => MPTCE f a where
  update = fmap
```

Multi-parameter type classes - When

The use of multi-parameter type classes can lead to ambiguity during type checking

This ambiguity can be resolved with Functional Dependencies, we'll see more in that extensions info.

FlexibleContexts - What

Allow the use of complex constraints in class declaration contexts.

FlexibleContexts - Why

In Haskell 98 the context of a class declaration (which introduces superclasses) must be simple; that is, each predicate must consist of a class applied to type variables.

FlexibleContexts - How

```haskell
class (Monad m, Monad (t m)) => Transform t m where
  lift :: m a -> (t m) a

class FlexibleContextsExample a b where
  transformInto :: a -> b

instance Show a => FlexibleContextsExample a [Char] where
  transformInto = show

intoString :: FlexibleContextsExample a [Char] => a -> [Char]
intoString = transformInto
```

```
-- src\FlexibleContexts.hs:23:15: error:
--     * Non type-variable argument
--         in the constraint: FlexibleContextsExample a [Char]
--       (Use FlexibleContexts to permit this)
--     * In the type signature:
--         intoString :: FlexibleContextsExample a [Char] => a ->
[Char]
--    |
-- 23 | intoString :: FlexibleContextsExample a [Char] => a ->
[Char]
--    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

ConstrainedClassMethods - What

Allows the definition of further constraints on individual class methods.

ConstrainedClassMethods - Why

The restriction is a pretty stupid one in the first place

Implied by MultiParamTypeClasses

ConstrainedClassMethods - How

```
class Seq s a where
  fromList :: [a] -> s a
  elem     :: Eq a => a -> s a -> Bool

-- src\ConstrainedClassMethods.hs:19:3: error:
--     * Constraint `Eq a' in the type of `ConstrainedClassMethods.elem'
--         constrains only the class type variables
--       Enable ConstrainedClassMethods to allow it
--     * When checking the class method:
--         ConstrainedClassMethods.elem :: forall (s :: * -> *) a.
--                                         (Seq s a, Eq a) =>
--                                         a -> s a -> Bool
--       In the class declaration for `Seq'
--    |
-- 19 |   elem     :: Eq a => a -> s a -> Bool
--    |   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

DefaultSignatures - What

Allows the definition of default method signatures in class definitions.

DefaultSignatures - Why

Pairs extremely well with Generics programming

The Haskell2010 Report allows for default class methods, but not specialization of such methods

DefaultSignatures - How

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#generic-defaults

```
class Serialize a where
  put :: a -> [Bin]
  default put :: (Generic a, GSerialize (Rep a)) => a -> [Bin]
  put = gput . from
instance (Serialize a) => Serialize (UserTree a)

-- src\DefaultSignatures.hs:41:3: error:
--     Unexpected default signature:
--       default put :: (Generic a, GSerialize (Rep a)) => a -> [Bin]
--     Use DefaultSignatures to enable default signatures
--    |
-- 41 |   default put :: (Generic a, GSerialize (Rep a)) => a -> [Bin]
--    |   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  -- 4) Cautionary tales, if applicable
```

NullaryTypeClasses - What

Allows the use definition of type classes with no parameters.

This extension has been replaced by MultiParamTypeClasses.

NullaryTypeClasses - Why

Used in code documentation mostly

NullaryTypeClasses - How

```
class RiemannHypothesis where
  assumeRH :: a -> a

-- Deterministic version of the Miller test
-- correctness depends on the generalised Riemann hypothesis
isPrime :: RiemannHypothesis => Integer -> Bool
isPrime n = assumeRH undefined -- (...)

-- The type signature of isPrime informs users that its correctness depends on an unproven conjecture. If the function
is used, the user has to acknowledge the dependence with:

instance RiemannHypothesis where
  assumeRH = id
```

FunctionalDependencies - What

Allow use of functional dependencies in class declarations.

Implies: MultiParamTypeClasses

A functional dependency is a way to partner with the compiler to say that a specified type in a context will uniquely specify another type in the same context.

FunctionalDependencies - Why

By including dependencies in a class declaration, we provide a mechanism for the programmer to specify each multiple parameter class more precisely. The compiler, on the other hand, is responsible for ensuring that the set of instances that are in scope at any given point in the program is consistent with any declared dependencies.

“We say that an expression e has an ambiguous type if, in its type ∀ u. cx ⇒ t, there is a type variable u in u that occurs in cx but not in t. Such types are invalid.” - Haskel2010

More generally, we need only regard a type as ambiguous if it contains a variable on the left of the => that is not uniquely determined (either directly or indirectly) by the variables on the right.

FunctionalDependencies - How

```
class Collects e ce | ce -> e where
  empty  :: ce
  insert :: e -> ce -> ce
  member :: e -> ce -> Bool

-- src\FunctionalDependencies.hs:15:1: error:
--     * Fundeps in class `Collects'
--       (Enable FunctionalDependencies to allow fundeps)
--     * In the class declaration for `Collects'
--   |
-- 15 | class Collects e ce | ce -> e where
--   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Instance Extensions

- TypeSynonymInstances
- FlexibleInstances
- UndecidableInstances
- OverlappingInstances
- IncoherentInstances
- InstanceSigs
- OverloadedStrings
- OverloadedLabels
- OverloadedLists
- UndecidableSuperClasses

TypeSynonymInstances - What

Allow definition of type class instances for type synonyms.

TypeSynonymInstances - Why

Convenience

Just lets you define instances on synonyms for clarity

TypeSynonymInstances - How

```
newtype Foo = Foo String
type Fooo = Foo

instance Semigroup Fooo where
  (<>) (Foo x) (Foo y) = Foo (x <> y)
```

```
-- src\TypeSynonymInstances.hs:16:12: error:
--     * Illegal instance declaration for `Monoid Fooo'
--         (All instance types must be of the form (T t1 ... tn)
--          where T is not a synonym.
--          Use TypeSynonymInstances if you want to disable this.)
--     * In the instance declaration for `Monoid Fooo'
--    |
-- 16 |   instance Monoid Fooo where
--    |            ^^^^^^^^^^
```

FlexibleInstances - What

Allow definition of type class instances with arbitrary nested types in the instance head.

Implies TypeSynonymInstances

FlexibleInstances - Why

Allows arbitrary nesting of types in the instance head

FlexibleInstances - How

```haskell
class FlexibleInstancesExample a b where
  transformInto :: a -> b

instance Show a => FlexibleInstancesExample a [Char]
where
  transformInto = show

intoString :: FlexibleInstancesExample a [Char] => a -> [Char]
intoString = transformInto
```

```
-- src\FlexibleInstances.hs:15:20: error:
--     * Illegal instance declaration for
--         `FlexibleInstancesExample a [Char]'
--       (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
--         `FlexibleInstancesExample a [Char]'
--    |
-- 15 | instance Show a => FlexibleInstancesExample a [Char]
where
--    |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Detour: Instance Resolution

When unifying instances, the compiler wants to ensure that we don’t loop forever.

You can think of the rules as saying we always make the instance matching problem smaller.

Detour: Instance Resolution

1. ***The Paterson Conditions: for each class constraint*** `(C t1 ...tn)` ***in the context***
   1. ***No type variable has more occurrences in the constraint than in the head***
   2. ***The constraint has fewer constructors and variables (taken together and counting repetitions) than the head***
   3. ***The constraint mentions no type functions. A type function application can in principle expand to a type of arbitrary size, and so are rejected out of hand***
2. ***The Coverage Condition. For each functional dependency,*** $\langle tvs \rangle_{left}$ `->` $\langle tvs \rangle_{right}$***, of the class, every type variable in*** $S(\langle tvs \rangle_{right})$ ***must appear in*** $S(\langle tvs \rangle_{left})$***, where S is the substitution mapping each type variable in the class declaration to the corresponding type in the instance head.***

Detour: Instance Resolution Patterson 1

1. ***for each class constraint*** `(C t1 ...tn)` ***in the context***
    1. ***No type variable has more occurrences in the constraint than in the head***

```
instance Foo a a => Bar a where
```

Detour: Instance Resolution Patterson 2

1. ***for each class constraint*** `(C t1 ...tn)` ***in the context***
    1.
    2. ***The constraint has fewer constructors and variables (taken together and counting repetitions) than the head***

```
instance Show [Foo a] => Show (Foo a) where

  show = show . (:[])
```

Detour: Instance Resolution Patterson 3

1. ***for each class constraint*** `(C t1 ...tn)` ***in the context***
   1.
   2.
   3. ***The constraint mentions no type functions. A type function application can in principle expand to a type of arbitrary size, and so are rejected out of hand***

```
class Const a where
  type ExampleFam a :: * -> *
  toUnit :: a -> ()

instance Show (ExampleFam a Int) => Const a where
  toUnit = const ()
```

Detour: Instance Resolution Coverage Condition

1.
2. ***The Coverage Condition. For each functional dependency, $\langle tvs \rangle_{left}$ -> $\langle tvs \rangle_{right}$, of the class, every type variable in $S(\langle tvs \rangle_{right})$ must appear in $S(\langle tvs \rangle_{left})$, where S is the substitution mapping each type variable in the class declaration to the corresponding type in the instance head.***

```
class Mul a b c | a b -> c where
  (.*.) :: a -> b -> c

instance Mul Int Int a
```

UndecidableInstances - What

if you use the experimental extension UndecidableInstances, both the Paterson Conditions and the Coverage Condition are lifted. Termination is still ensured by having a fixed-depth recursion stack.

Relaxes the rules that guarantee instance resolution terminates

UndecidableInstances - Why

1) Class synonyms

```
class (C1 a, C2 a, C3 a) => C a where { }
```

2) Functional Dependencies

3) Type Families

UndecidableInstances - How

```haskell
class HasConverter a b | a -> b where
  convert :: a -> b

data Foo a = MkFoo a

instance (HasConverter a b,Show b) => Show (Foo a) where
  show (MkFoo value) = show (convert value)
```

```
src\UndecidableInstances.hs:36:10: error:
    * Variable `b' occurs more often
        in the constraint `HasConverter a b'
        than in the instance head `Show (Foo a)'
      (Use UndecidableInstances to permit this)
    * In the instance declaration for `Show (Foo a)'
   |
36 | instance (HasConverter a b,Show b) => Show (Foo a)
where
   |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

src\UndecidableInstances.hs:36:10: error:
    * Variable `b' occurs more often
        in the constraint `Show b' than in the instance head
`Show (Foo a)'
      (Use UndecidableInstances to permit this)
    * In the instance declaration for `Show (Foo a)'
   |
36 | instance (HasConverter a b,Show b) => Show (Foo a)
where
   |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

UndecidableInstances - When

```
class Mul a b c | a b -> c where
  (.*.) :: a -> b -> c

instance Mul Int Int Int where (.*.) = (*)
instance Mul Int Float Float where x .*. y = fromIntegral x * y
instance Mul a b c => Mul a [b] [c] where x .*. v = map (x.*.) v

f = \ b x y -> if b then x .*. [y] else y
```

```
src\UndecidableInstances.hs:54:26: error:
    * Reduction stack overflow; size = 201
      When simplifying the following type: Mul a0 [[c]] [c]
      Use -freduction-depth=0 to disable this check
      (any upper bound you could choose might fail
unpredictably with
       minor updates to GHC, so disabling the check is
recommended if
       you're sure that type checking should terminate)
    * In the expression: x .*. [y]
      In the expression: if b then x .*. [y] else y
      In the expression: \ b x y -> if b then x .*. [y] else y
   |
54 | f = \ b x y -> if b then x .*. [y] else y
   |                ^^^^^^^^^
```

OverlappingInstances - What

Deprecated extension to weaken checks intended to ensure instance resolution termination.

Top level extension is deprecated but can still be used inline on definitions

OverlappingInstances - Why

Conveniently define default behavior with exceptions for special cases of types.

OverlappingInstances - How

```
instance Show a => Show [a] where

instance {-# OVERLAPPING #-} Show [Char] where

instance {-# OVERLAPPING #-}, {-# OVERLAPPABLE #-}, {-# OVERLAPS #-}
```

Conveniently define default behavior with exceptions for special cases of types.

OVERLAPS = OVERLAPPING + OVERLAPPABLE

OVERLAPPING = Overwrite other instances

OVERLAPPABLE = Can be overwritten

IncoherentInstances - What

Deprecated extension to weaken checks intended to ensure instance resolution termination.

Top level extension is deprecated but can still be used inline on definitions

IncoherentInstances - Why

allow[s] more than one instance to match irrespective of whether there is a most specific one.

Force instances in a particular module despite possible changes.

Cause early commit to a particular instance in module compilation.

Constraint Searching Rules (1/2)

Find all instances I that match the target constraint; that is, the target constraint is a substitution instance of I. These instance declarations are the candidates.

Eliminate any candidate I_x for which both of the following hold:

> ***There is another candidate I_y that is strictly more specific; that is, I_y is a substitution instance of I_x but not vice versa.***
>
> ***Either I_x is overlappable, or I_y is overlapping. (This "either/or" design, rather than a "both/and" design, allow a client to deliberately override an instance from a library, without requiring a change to the library.)***

Constraint Searching Rules (2/2)

If exactly one non-incoherent candidate remains, select it. If all remaining candidates are incoherent, select an arbitrary one. Otherwise the search fails (i.e. when more than one surviving candidate is not incoherent).

If the selected candidate (from the previous step) is incoherent, the search succeeds, returning that candidate.

If not, find all instances that unify with the target constraint, but do not match it. Such non-candidate instances might match when the target constraint is further instantiated. If all of them are incoherent, the search succeeds, returning the selected candidate; if not, the search fails.

IncoherentInstances - How

```
instance {-# OVERLAPPABLE #-} context1 => C Int b     where ...  -- (A)
instance {-# OVERLAPPABLE #-} context2 => C a   Bool  where ...  -- (B)
instance {-# OVERLAPPABLE #-} context3 => C a   [b]   where ...  -- (C)

f :: C Int [Int] -> Int
```

Overlapping and Incoherent - When

Warning

Overlapping instances must be used with care. They can give rise to incoherence (i.e. different instance choices are made in different parts of the program) even without `IncoherentInstances` . Consider:

Overlapping and Incoherent - When

```haskell
{-# LANGUAGE OverlappingInstances #-}
module Help where

  class MyShow a where
  myshow :: a -> String

  instance MyShow a => MyShow [a] where
  myshow xs = concatMap myshow xs

  showHelp :: MyShow a => [a] -> String
  showHelp xs = myshow xs
```

```haskell
{-# LANGUAGE FlexibleInstances, OverlappingInstances #-}
module Main where
  import Help

  data T = MkT

  instance MyShow T where
  myshow x = "Used generic instance"

  instance MyShow [T] where
  myshow xs = "Used more specific instance"

  main = do { print (myshow [MkT]); print (showHelp [MkT]) }
```

```
> "Used more specific instance"
> "Used generic instance"
```

Overlapping and Incoherent - When

Avoid if at all possible

Alternatives:

1. Add more class functions
2. Closed Type Families
3. Auxiliary classes with kind flags
4. Type Equality Analysis

InstanceSigs - What

Allow type signatures for members in instance definitions.

InstanceSigs - Why

Useful for helping scope type variables or for being explicit about documenting signatures of functions

InstanceSigs - How

```
class ContainsTriplicate a where
  tripUp :: b -> a -> (a, [b])

instance ContainsTriplicate a => ContainsTriplicate (Maybe a) where
  tripUp :: forall b. b -> Maybe a -> (Maybe a, [b])
  tripUp x Nothing = (Nothing, [])
  tripUp x (Just y) = (Just y, xs)
    where
      xs :: [b] -- Explicitly noting the type of our list here
      xs = [x,x,x]
```

```
-- src\InstanceSigs.hs:17:13: error:
--     * Illegal type signature in instance declaration:
--         tripUp :: forall b. b -> Maybe a -> (Maybe a, [b])
--       (Use InstanceSigs to allow this)
--     * In the instance declaration for `ContainsTriplicate (Maybe a)'
--    |
-- 17 |   tripUp :: forall b. b -> Maybe a -> (Maybe a, [b])
--    |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

OverloadedStrings - What

Enable overloaded string literals (e.g. string literals desugared via the IsString class).

OverloadedStrings - Why

Incredibly Convenient

OverloadedStrings - How

```haskell
data Configuration = Config {
    a :: Text
  , b :: ByteString
  , c :: String
  }

defaultConfig :: Configuration
defaultConfig = Config "a" "b" "c"
```

```
-- src\OverloadedStrings.hs:24:24: error:
--     * Couldn't match expected type `Text' with actual type `[Char]'
--     * In the first argument of `Config', namely `"a"'
--       In the expression: Config "a" "b" "c"
--       In an equation for `defaultConfig':
--           defaultConfig = Config "a" "b" "c"
--   |
-- 24 | defaultConfig = Config "a" "b" "c"
--   |                        ^^^

-- src\OverloadedStrings.hs:24:28: error:
--     * Couldn't match expected type `ByteString'
--                   with actual type `[Char]'
--     * In the second argument of `Config', namely `"b"'
--       In the expression: Config "a" "b" "c"
--       In an equation for `defaultConfig':
--           defaultConfig = Config "a" "b" "c"
--   |
-- 24 | defaultConfig = Config "a" "b" "c"
--   |                            ^^^
```

OverloadedLabels - What

Enable use of the #foo overloaded label syntax.

OverloadedLabels - Why

Not much utility today outside of special circumstances

The intention is for IsLabel to be used to support overloaded record fields and perhaps anonymous records. Thus, it may be given instances for base datatypes (in particular (->)) in the future.

OverloadedLabels - How

```haskell
import GHC.OverloadedLabels (IsLabel(..))
import GHC.TypeLits (Symbol)

data Label (l :: Symbol) = Get

class Has a l b | a l -> b where
  from :: a -> Label l -> b

data Point = Point Int Int deriving Show

instance Has Point "x" Int where from (Point x _) _ = x
instance Has Point "y" Int where from (Point _ y) _ = y
```

```haskell
instance Has a l b => IsLabel l (a -> b) where
  fromLabel x = from x (Get :: Label l)

example = #x (Point 1 2)

-- src\OverloadedLabels.hs:32:11: error: parse error on
input `#'
--    |
-- 32 | example = #x (Point 1 2)
--    |           ^
```

OverloadedLists - What

Enable overloaded list syntax (e.g. desugaring of lists via the IsList class).

OverloadedLists - Why

This extension allows programmers to use the list notation for construction of structures like: Set, Map, IntMap, Vector, Text and Array.

OverloadedLists - How

```
type RoomNumber = Int
guests :: Map String RoomNumber
guests = [("John Doe", 108), ("Jane Smith", 419)]

-- src\OverloadedLists.hs:15:10: error:
--     * Couldn't match expected type `Map String RoomNumber'
--                   with actual type `[([Char], Integer)]'
--     * In the expression: [("John Doe", 108), ("Jane Doe", 419)]
--       In an equation for `guests':
--           guests = [("John Doe", 108), ("Jane Doe", 419)]
--    |
-- 15 | guests = [("John Doe", 108), ("Jane Doe", 419)]
--    |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

UndecidableSuperClasses - What

Allow all superclass constraints, including those that may result in non-termination of the typechecker.

UndecidableSuperClasses - Why

Provides more loosening of termination checks for superclass constraints

Cyclic constraints could terminate, but GHC isn’t able to check if they will. If you know it will terminate, you can lift the cycle restraints.

UndecidableSuperClasses - How

Moreover genuninely-recursive superclasses are sometimes useful. Here's a real-life example (Trac #10318)
Here the superclass cycle does terminate but it's not entirely straightforward to see that it does.

```
class (Frac (Frac a) ~ Frac a,
       Fractional (Frac a),
       IntegralDomain (Frac a))
   => IntegralDomain a where
  type Frac a :: Type

-- src\UndecidableSuperClasses.hs:20:1: error:
--     * Superclass cycle for `IntegralDomain'
--         one of whose superclasses is `IntegralDomain'
--       Use UndecidableSuperClasses to accept this
--     * In the class declaration for `IntegralDomain'
--    |
-- 20 | class (Frac (Frac a) ~ Frac a,
--    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Questions?

https://github.com/jkeuhlen

@jkeuhlen on FP Slack

jkeuhlen.com

Resources

- https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#class-and-instances-declarations
- https://www.haskell.org/definition/haskell2010.pdf
- https://www.microsoft.com/en-us/research/wp-content/uploads/1997/01/multi.pdf