Type Classes in Haskell

Tom Schrijvers

KU LEUVEN

Leuven Haskell User Group

Haskell Research Team

Partners

Microsoft Research Cambridge

UNIVERSITY OF OXFORD

Monads

Type Classes

GHC

Folds

Pattern Matching

Equational Reasoning

DSLs

Advanced Types

Adhoc Overloading

Example: Addition

Different Implementations for different types

```
Int -> Int -> Int

Float -> Float -> Float

Vector -> Vector -> Vector

...
```

How do we name the implementations?

Different Names for Different Implementations

e.g. Ocaml

```
+   : int -> int -> int
```

```
+. : float -> float -> float
```

😟

- remember more names
- common concept / properties

Same Name for Different Implementations

e.g. Java

```
+   : int -> int -> int
```

```
+ : float -> float -> float
```

adhoc overloading

operator overloading

adhoc polymorphism

- hard-wired in the language
- not user-extensible

Type Classes for Adhoc Overloading

Type Classes

"How to make ad-hoc polymorphism less ad hoc"

Philip Wadler and Stephen Blott. 1989.

Example: Equality

```
(==) :: a -> a -> Bool
```

equality is adhoc
overloaded

```
> "hello" == "world"
False

> 2 == (1+1)
True
```

Example: Equality

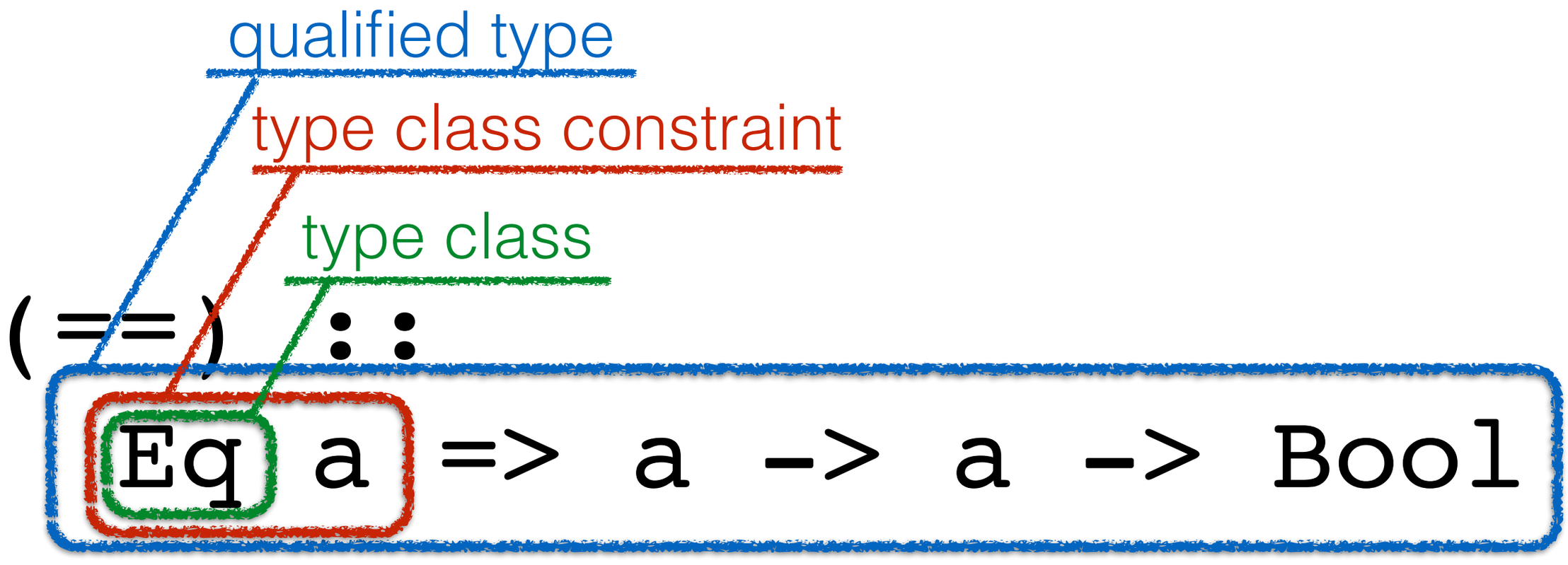
```
(==) :: a -> a -> Bool
```

not every type supports equality

```
> id == (\x -> x)

    No instance for (Eq (a -> a))
      arising from a use of ‘==’
    In the expression: id == (\ x -> x)
```

Constraint Polymorphism



type **a** must have an implementation of equality

Type Class Declaration

```
class Eq a where
  (==) :: a -> a -> Bool
```

method

Type Class Instantiation

```
data Color = Red | Blue
```

```
> Red == Red
True
> Red == Blue
False
```

Using Methods

```
isRed :: Color -> Bool
isRed c  =  c == Red
```

ok because
`Eq Color` instance
exists

Constraint Polymorphic Use

most general type?

```
same (x,y)  =  x == y
```

type class constraints propagate!

Multiple Methods

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Type Class Instantiation

```
data Color = Red | Blue

instance Eq Color where
  Red  == Red   =  True
  Blue == Blue  =  True
  _    == _     =  False

  x    /= y     =  not (x == y)
```

Default Implementations

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
x /= y = not (x == y)
```

Default Implementation

```
data Color = Red | Blue

instance Eq Color where
  Red  == Red   =  True
  Blue == Blue  =  True
  _    == _     =  False
  _

  x /= y        =  not (x == y)
```

Default Implementations

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

```
class Eq a where
```

The `Eq` class defines equality (`==`) and inequality (`/=`). All the basic datatypes exported by the Prel
`Eq`, and `Eq` may be derived for any datatype whose constituents are also instances of `Eq`.

Minimal complete definition: either `==` or `/=`.

Minimal complete definition

```
(==) | (/=)
```

Instance Contexts

instance context

```
instance Eq a => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x==y && xs==ys
  _      ==  _     =  False
```

Instance Contexts

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,x2) == (y1,y2)
     =  x1 == y1 && x2 == y2
```

Subclassing

super class

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

every instance of **`Ord`**
must also have
an instance of **`Eq`**

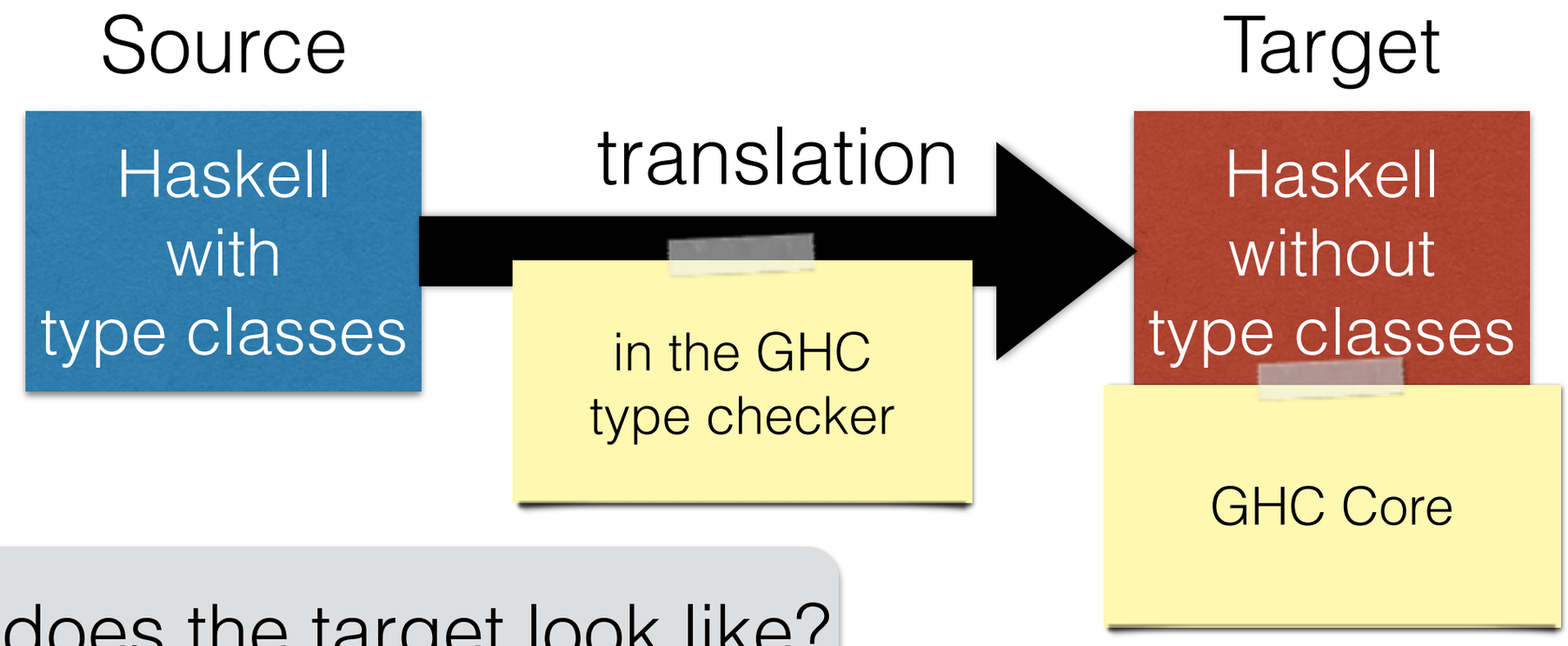
```
r' :: Ord a => a -> a -> Bool
r' x y  =  x == y
```

The `Num` Class

```
class Num a where
  (+)                 :: a -> a -> a
  (-)                 :: a -> a -> a
  (*)                 :: a -> a -> a
  negate              :: a -> a
  abs                 :: a -> a
  signum              :: a -> a
  fromInteger         :: Integer -> a
```

Dictionary-Passing Translation

Implementation



What does the target look like?

How is the target obtained?

Dictionary Representation

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
data Eq a =
  Eq { (==) :: a -> a -> Bool
     , (/=) :: a -> a -> Bool
     }
```

(function) dictionary

Instance Translation

```
instance Eq () where
  (==) = \x y -> True
  (/=) = \x y -> False
```

```
eqUnit :: Eq ()
eqUnit =
  Eq { (==) = \x y -> True
     , (/=) = \x y -> False
     }
```

Call Translation

```
p :: () -> () -> Bool
p x y  =  x == y
```

```
p :: () -> () -> Bool
p x y  =  (==) eqUnit x y
```

typically followed by inlining

Instance Translation

```
instance (Eq a,Eq b) => Eq (a,b) where
  (==) = \(x1,y1) (x2,y2)
         -> x1 == x2 && y1 == y2
```



```
eqPair :: Eq a -> Eq b -> Eq (a,b)
eqPair da db =
  Eq { (==) =
         \(x1,y1) (x2,y2)
           -> (==) d1 x1 x2 &&
              (==) d2 y1 y2
     , … }
```

Call Translation

```
q :: ((),()) -> ((),()) -> Bool
q x y  =  x == y
```



```
p :: ((),()) -> ((),()) -> Bool
p x y  =  (==) d x y
       where
         d = eqPair eqUnit eqUnit
```

Constraint Polymorphic Signature Translation

```
r :: Eq a => a -> a -> Bool
r x y  =  x == y
```

```
r :: Eq a -> a -> a -> Bool
r d x y  =  (==) d x y
```

Constraint Polymorphic Signature Translation

```
s :: Eq a => (a,a) -> (a,a) -> Bool
s x y  =  x == y
```

```
s :: Eq a -> (a,a) -> (a,a) -> Bool
s d x y  =  (==) d' x y
          where
            d' = eqPair d d
```

Super Class Translation

```
class Eq a => Ord a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
data Ord a =
  Ord { super :: Eq a
      , …
      }
```

Super Class Access

```
r' :: Ord a => a -> a -> Bool
r' x y  =  x == y
```



```
r' :: Ord a -> a -> a -> Bool
r' d x y  =  (==) d' x y
          where
            d' = super d
```

Resolution

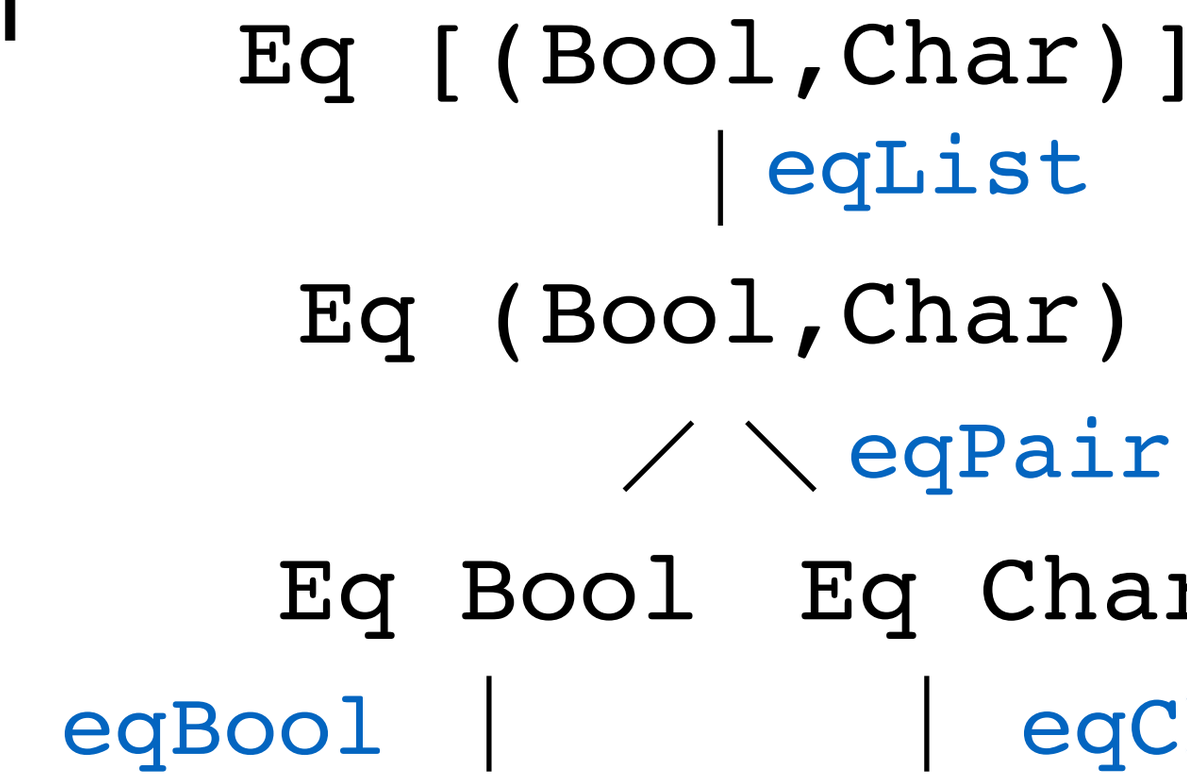
Resolution Process

```
[(True,'a')] == [(False,'b')]
```

```
(==) ? [] [(True,'a')] [(False,'b')]
```

Eq [(Bool,Char)]

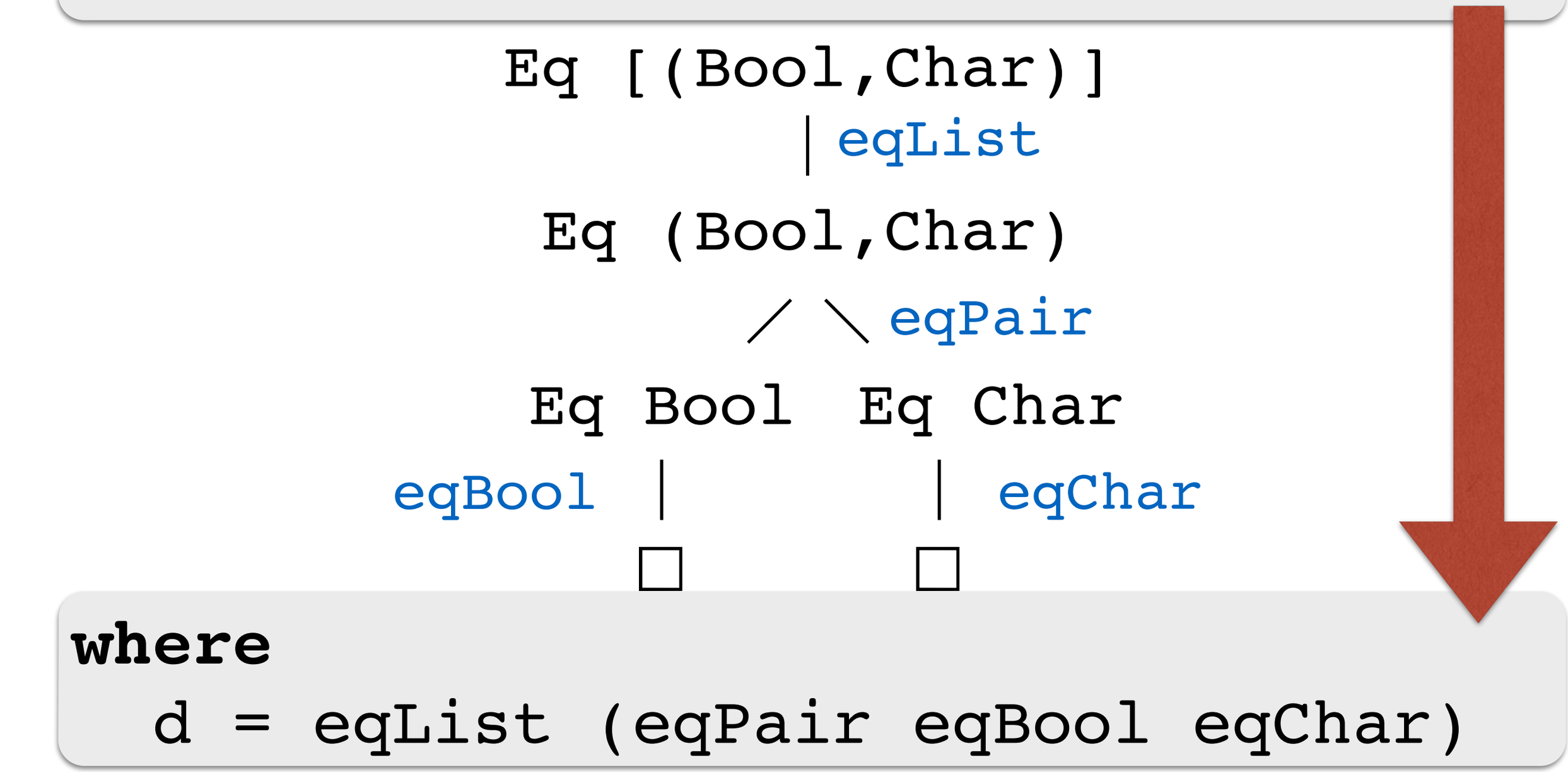
Resolution Process




```
instance Eq a                 => Eq [a]
instance (Eq a, Eq b) => Eq (a,b)
instance                         Eq Bool
instance                         Eq Char
…
```

Resolution Process

```
(==) d [] [(True,'a')] [(False,'b')]
```




```
where
  d = eqList (eqPair eqBool eqChar)
```

Summary

Type Classes

★Systematic approach to adhoc overloading

★implemented by means of dictionary passing

★where resolution assembles the dictionaries.

LANGUAGE Pragmas

```
MultiParamTypeClasses

FunctionalDependencies

FlexibleContexts

FlexibleInstances

UndecidableInstances

OverlappingInstances

IncoherentInstances
```

non-terminating resolution possible

incoherent resolution possible

incoherent resolution likely

Join the Google Group

Leuven Haskell User Group

Join the Meetup

Leuven Haskell User Group