

# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**

[andrea@di.unipi.it](mailto:andrea@di.unipi.it)

<http://pages.di.unipi.it/corradini/>

Course pages:

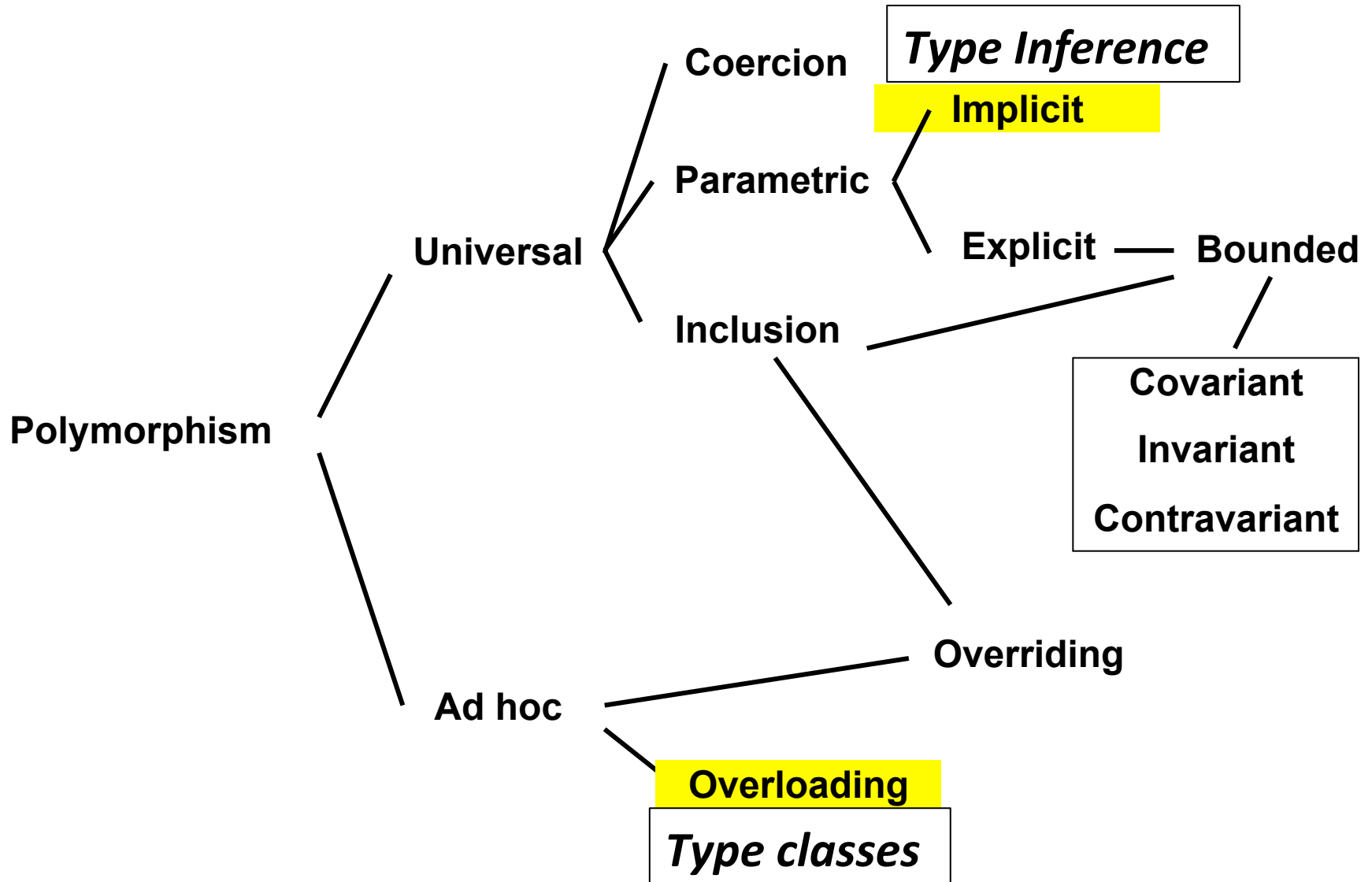
<http://pages.di.unipi.it/corradini/Didattica/AP-18/>

***AP-2018-20: Type Classes in Haskell***

# Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
  - Type Classes
  - Monads
- Exceptions

# Polymorphism in Haskell



# Ad hoc polymorphism: **overloading**

- Present in all languages, at least for built-in arithmetic operators: +, \*, -, ...
- Sometimes supported for user defined functions (Java, C++, ...)
- C++, Haskell allow overloading of primitive operators
- The code to execute is determined by the **type of the arguments**, thus
  - **early binding** in statically typed languages
  - **late binding** in dynamically typed languages

# Overloading: an example

- Function for squaring a number:  
`sqr(x) { return x * x; }`
- Typed version (like in C) :  
`int sqr(int x) { return x * x; }`
- Multiple versions for different types:  
`int sqrInt(int x) { return x * x; }`  
`double sqrDouble(double x) { return x * x; }`
- Overloading (Java, C++):  
`int sqr(int x) { return x * x; }`  
`double sqr(double x) { return x * x; }`
- But which type can be inferred by ML/Haskell?  
`> sqr x = x * x`

# Overloading besides arithmetic

- Some functions are "fully polymorphic"

```
length :: [w] -> Int
```

- Many useful functions are less polymorphic

```
member :: [w] -> w -> Bool
```

- Can list membership work for any type?
  - No! Only for types **w** that support equality.

```
sort :: [w] -> [w]
```

- Can list sorting work for any type?
  - No! Only for types **w** that support ordering.

# Overloading Arithmetic, Take 1

- Allow functions containing overloaded symbols to define multiple functions:

```
square x = x * x          -- legal
-- Defines two versions:
-- Int -> Int and Float -> Float
```

- But consider:

```
squares (x,y,z) =
    (square x, square y, square z)
-- There are 8 possible versions!
```

- Approach not widely used because of exponential growth in number of versions.

# Overloading Arithmetic, Take 2

- Basic operations such as `+` and `*` can be overloaded, but not functions defined from them

```
3 * 3                -- legal
3.14 * 3.14          -- legal
square x = x * x     -- Int -> Int
square 3              -- legal
square 3.14          -- illegal
```

- Standard ML** uses this approach.
- Not satisfactory: Programmer cannot define functions that implementation might support



# Overloading Equality, Take 1

- Equality defined only for types that admit equality: types not containing **function** or **abstract types**.

```
3 * 3 == 9           -- legal
'a' == 'b'           -- legal
\x->x == \y->y+1       -- illegal
```

- Overload equality like arithmetic ops + and \* in SML.
- But then we can't define functions using '==':

```
member [] y          = False
member (x:xs) y = (x==y) || member xs y

member [1,2,3] 3      -- ok if default is Int
member "Haskell" 'k'  -- illegal
```

- Approach adopted in first version of SML.

# Overloading Equality, Take 2

- Make type of equality fully polymorphic

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

- Type of list membership function

```
member :: [a] -> a -> Bool
```

- **Miranda** used this approach.
  - Equality applied to a **function** yields a runtime error
  - Equality applied to an **abstract type** compares the underlying representation, which violates abstraction principles

# Overloading Equality, Take 3

- Make equality polymorphic **in a limited way**:

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

where  $a(==)$  is type variable restricted to **types with equality**

- Now we can type the member function:

```
member :: a(==) -> [a(==)] -> Bool
member 4      [2,3] :: Bool
member 'c'    ['a', 'b', 'c'] :: Bool
member (\y -> y*2) [\x -> x, \x -> x+2] -- type error
```

- Approach used in SML today, where the type  $a(==)$  is called an **eqtype variable** and is written **"a** (while normal type variables are written **'a** )

# Type Classes

- Type classes solve these problems
  - Provide concise types to describe overloaded functions, so no exponential blow-up
  - Allow users to define functions using overloaded operations, eg, `square`, `squares`, and `member`
  - Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged built-ins
  - Generalize ML's `eqtypes` to arbitrary types
  - Fit within `type inference framework`

# Intuition

- A function to sort lists can be passed a **comparison operator** as an argument:

```
qsort:: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
                  ++ [x] ++
                  qsort cmp (filter (not.cmp x) xs)
```

- This allows the function to be parametric
- We can built on this idea ...

# Intuition (continued)

- Consider the “overloaded” parabola function

```
parabola x = (x * x) + x
```

- We can rewrite the function to take the operators it contains as an argument

```
parabola' (plus, times) x = plus (times x x) x
```

- The extra parameter is a “dictionary” that provides implementations for the overloaded ops.
- We have to rewrite all calls to pass appropriate implementations for plus and times:

```
y = parabola' (intPlus, intTimes) 10  
z = parabola' (floatPlus, floatTimes) 3.14
```

# Systematic programming style

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p

get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t

-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus  = get_plus dict
                  times = get_times dict
                  in plus (times x x) x
```

**Type class declarations**  
will generate Dictionary  
type and selector  
functions

# Systematic programming style

Type class **instance declarations**  
produce instances of the Dictionary

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Dictionary construction
intDict    = MkMathDict intPlus    intTimes
floatDict  = MkMathDict floatPlus  floatTimes

-- Passing dictionaries
y = parabola intDict    10
z = parabola floatDict  3.14
```

Compiler will add a dictionary  
parameter and rewrite the body as  
necessary



# Type Class Design Overview

- **Type class declarations**
  - Define a set of operations, give the set a name
  - Example: `Eq a` type class
    - operations `==` and `\=` with `type a -> a -> Bool`
- **Type class instance declarations**
  - Specify the implementations for a particular type
  - For `Int` instance, `==` is defined to be integer equality
- **Qualified types (or Type Constraints)**
  - Concisely express the operations required on otherwise polymorphic type

```
member :: Eq w => w -> [w] -> Bool
```

“for all types  $w$  that support the `Eq` operations”

# Qualified Types

```
Member :: Eq w => w -> [w] -> Bool
```

If a function works for every type with particular properties, the type of the function says just that:

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a  -> String
square    :: Num n  => n  -> n
squares   :: (Num t, Num t1, Num t2) =>
              (t, t1, t2) -> (t, t1, t2)
```

Otherwise, it must work for any type

```
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

Works for any type  
'n' that supports  
the Num operations

# Type Classes

```
square :: Num n => n -> n
square x = x*x
```

```
class Num a where
  (+)      :: a -> a -> a
  (*)      :: a -> a -> a
  negate  :: a -> a
  ...etc...
```

```
instance Num Int where
  a + b      = intPlus  a b
  a * b      = intTimes a b
  negate a   = intNeg  a
  ...etc...
```

The class declaration  
says what the Num  
operations are

An instance  
declaration for a  
type T says how the  
Num operations are  
implemented on T's

```
intPlus  :: Int -> Int -> Int
intTimes :: Int -> Int -> Int
etc, defined as primitives
```

# Compiling Overloaded Functions

When you write this...

```
square :: Num n => n -> n  
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n  
square d x = (*) d x x
```

The “Num n =>” turns into an extra value argument to the function. It is a value of data type Num n and it represents a dictionary of the required operations.

A value of type (Num n) is a dictionary of the Num operations for type n

# Compiling Type Classes

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
class Num n where
  (+)      :: n -> n -> n
  (*)      :: n -> n -> n
  negate  :: n -> n
  ...etc...
```

```
data Num n
  = MkNum (n -> n -> n)
          (n -> n -> n)
          (n -> n)
  ...etc...

...
(*) :: Num n -> n -> n -> n
(*) (MkNum _ m _ ...) = m
```

The class decl translates to:

A data type decl for Num  
A selector function for each class operation

A value of type (Num n) is a dictionary of the Num operations for type n

# Compiling Instance Declarations

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b      = intPlus  a b
  a * b      = intTimes a b
  negate a   = intNeg a
  ...etc...
```

```
dNumInt :: Num Int
dNumInt = MkNum intPlus
              intTimes
              intNeg
              ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

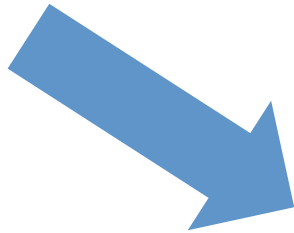
A value of type (Num n) is a dictionary of the Num operations for type n

# Implementation Summary

- The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: **the dictionary**.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
- The compiler converts each **type class declaration** into a **dictionary type declaration** and a set of **selector functions**.
- The compiler converts each **instance declaration** into a **dictionary** of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a dictionary. **It uses the static, qualified type of the function to select the dictionary.**

# Functions with Multiple Dictionaries

```
squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares(x,y,z) = (square x, square y, square z)
```



Note the concise type for the squares function!

```
squares :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
squares (da,db,dc) (x, y, z) =
    (square da x, square db y, square dc z)
```

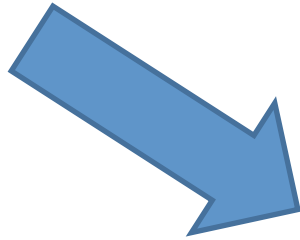
Pass appropriate dictionary on to each square function.



# Compositionality

Overloaded functions can be defined from other overloaded functions:

```
sumSq :: Num n => n -> n -> n  
sumSq x y = square x + square y
```



```
sumSq :: Num n -> n -> n -> n  
sumSq d x y = (+) d (square d x)  
              (square d y)
```

Extract addition  
operation from d

Pass on d to square

# Compositionality

Build compound instances from simpler ones:

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

instance Eq Int where
    (==) = intEq      -- intEq primitive equality

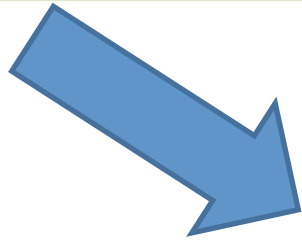
instance (Eq a, Eq b) => Eq (a,b)
    (u,v) == (x,y)      = (u == x) && (v == y)

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

# Compound Translation

Build compound instances from simpler ones.

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



```
data Eq = MkEq (a->a->Bool)           -- Dictionary type
(==) (MkEq eq) = eq                    -- Selector
dEqList :: Eq a -> Eq [a]              -- List Dictionary
dEqList d = MkEq eql
  where
    eql [] [] = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _ _ = False
```

# Many Type Classes

- **Eq**: equality
- **Ord**: comparison
- **Num**: numerical operations
- **Show**: convert to string
- **Read**: convert from string
- **Testable**, **Arbitrary**: testing.
- **Enum**: ops on sequentially ordered types
- **Bounded**: upper and lower values of a type
- Generic programming, reflection, monads, ...
- And many more.

# Subclasses

- We could treat the Eq and Num type classes separately

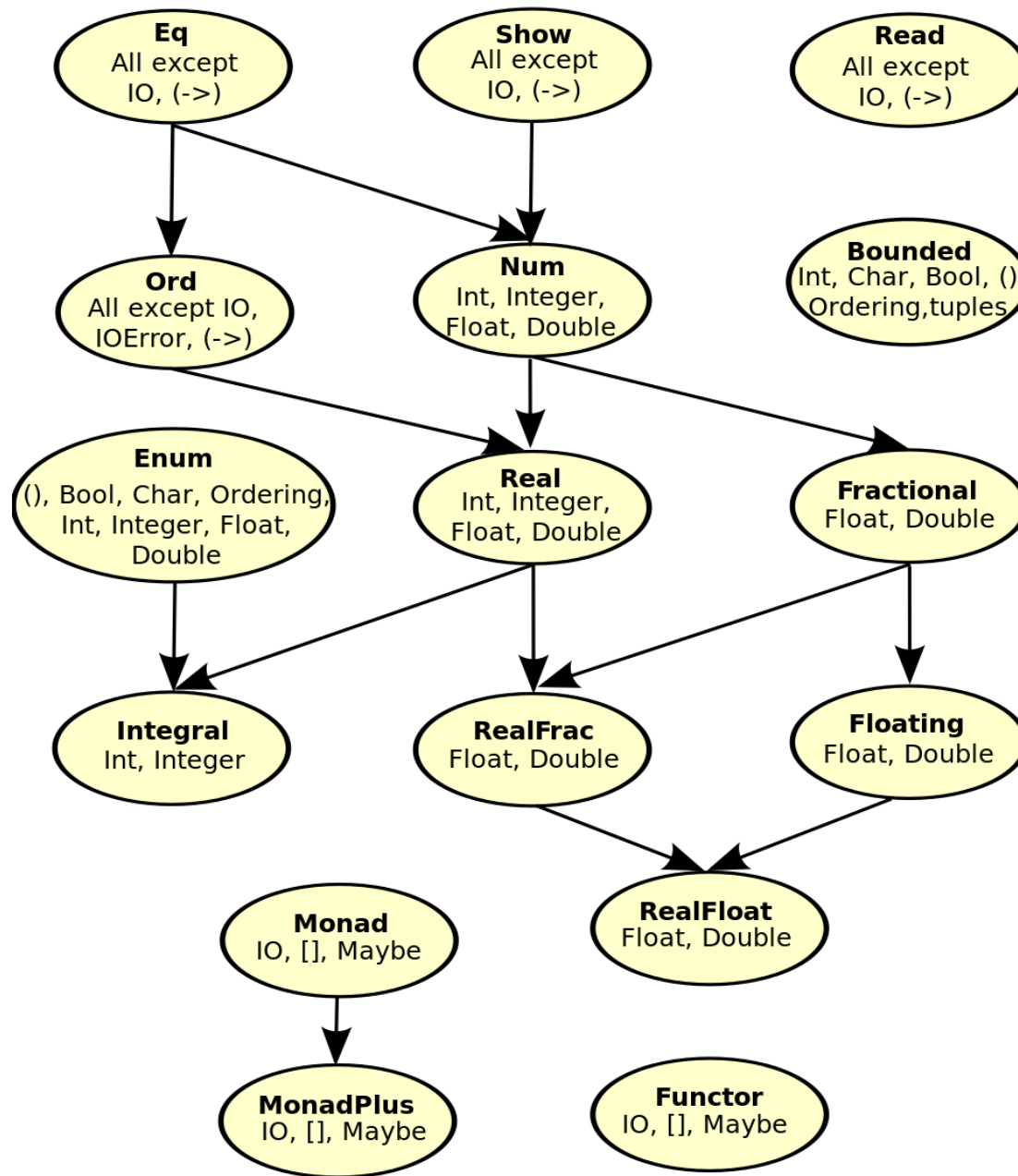
```
memsq :: (Eq a, Num a) => a -> [a] -> Bool
memsq x xs = member (square x) xs
```

- But we expect any type supporting Num to also support Eq
- A subclass declaration expresses this relationship:

```
class Eq a => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
```

- With that declaration, we can simplify the type of the function

```
memsq :: Num a => a -> [a] -> Bool
memsq x xs = member (square x) xs
```



# Default Methods

- Type classes can define “default methods”

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

- Instance declarations can override default by providing a more specific definition.

# Deriving

- For **Read**, **Show**, **Bounded**, **Enum**, **Eq**, and **Ord**, the compiler can generate instance declarations automatically

```
data Color = Red | Green | Blue
    deriving (Show, Read, Eq, Ord)
```

```
Main>:t show
show :: Show a => a -> String
Main> show Red
"Red"
Main> Red < Green
True
Main>:t read
read :: Read a => String -> a
Main> let c :: Color = read "Red"
Main> c
Red
```

- Ad hoc* : derivations apply only to types where derivation code works



# Numeric Literals

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ...

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded.  
`1 :: (Num a) => a`

"1" means  
"fromInteger 1"

## Advantages:

- Numeric literals can be interpreted as values of any appropriate numeric type
- Example: 1 can be an Integer or a Float or a user-defined numeric type.

# Type Inference with overloading

- In presence of overloading (Type Classes), type inference infers a **qualified type**  $Q \Rightarrow T$ 
  - $T$  is a Hindley Milner type, inferred as usual
  - $Q$  is set of type class predicates, called a **constraint**
- Consider the example function:

```
example z xs =  
  case xs of  
    []      -> False  
    (y:ys) -> y > z || (y==z && ys == [z])
```

- Type  $T$  is  $a \rightarrow [a] \rightarrow \text{Bool}$
- Constraint  $Q$  is  $\{ \text{Ord } a, \text{Eq } a, \text{Eq } [a] \}$

Ord a because  $y > z$   
Eq a because  $y == z$   
Eq [a] because  $ys == [z]$

# Simplifying Type Constraints

- Constraint sets  $Q$  can be simplified:
  - Eliminate duplicates
    - $(\text{Eq } a, \text{Eq } a)$  simplifies to  $\text{Eq } a$
  - Use an **instance declaration**
    - If we have  $\text{instance Eq } a \Rightarrow \text{Eq } [a]$ ,  
then  $(\text{Eq } a, \text{Eq } [a])$  simplifies to  $\text{Eq } a$
  - Use a **class declaration**
    - If we have  $\text{class Eq } a \Rightarrow \text{Ord } a$  where ...,  
then  $(\text{Ord } a, \text{Eq } a)$  simplifies to  $\text{Ord } a$
- Applying these rules,
  - $(\text{Ord } a, \text{Eq } a, \text{Eq } [a])$  simplifies to  $\text{Ord } a$

# Type Inference with overloading

- Putting it all together:

```
example z xs =  
  case xs of  
    []      -> False  
    (y:ys) -> y > z || (y==z && ys ==[z])
```

- $T = a \rightarrow [a] \rightarrow \text{Bool}$
- $Q = (\text{Ord } a, \text{Eq } a, \text{Eq } [a])$
- $Q$  simplifies to  $\text{Ord } a$
- `example :: Ord a => a -> [a] -> Bool`

# Detecting Errors

- Errors are detected when predicates are known not to hold:

```
Prelude> 'a' + 1
<interactive>:33:1: error:
  • No instance for (Num Char) arising from a use of '+'
  • In the expression: 1 + 'a'
    In an equation for `it`: it = 1 + 'a'
```

```
Prelude> (\x -> x)
<interactive>:34:1: error:
  • No instance for (Show (p0 -> p0)) arising from a use of `print'
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it
```