Types and Type Classes

ECS713

Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary
University of London

Week 5 Quiz

- https://qmplus.qmul.ac.uk/mod/quiz/view.php?id=1775351

- During lab this week: Friday 29th October: 14.05 - 15.55
- See web page for coverage.
- Open book.

Week 5: Contents

- Review: **type** and **data**
- Records
- Type classes
- Built-in classes: Eq, Ord, Num, Show, Read
- Creating new type classes

Types: Record notation
- understand how to use Haskell's record notation for user defined types Week 5/Task 1
Types: Polymorphism
- define polymorphic type definitions, and write type definitions that use polymorphism Week 5/Task 1
Type Classes: Built-in type classes
- familiar with Haskell's "built-in" types classes: Eq, Ord, Show, Read and Enum Week 5/Task 1
Type Classes: User defined type classes
- know how to define a new type class Week 5/Task 1
- use sub-classing to define a new type class that is a sub-class of an existing type class Week 5/Task 1
Type Classes: Making a type an instance of a type class
- know how to use the "deriving" keyword to make a new type an instance of the basic type classes Week 5/Task 1
- know how to use the "instance" keyword to make a new type an instance of a type class Week 5/Task 1

- be able to use Haskell "record" notation
- know what "polymorphism" means, and identify uses of it in Haskell
- know what a "type class" is, and can give examples of built-in Haskell type classes
- be able to define a new type class
- be able to make an existing type an instance of a given type class

Review: type and data

Type Synonyms

Use **type** to give a new name to an existing type

```
type String = [Char]
type Position = (Int, Int)
type Distance = Int
```


Makes typing more readable:

```
toUpperString :: String -> String
translate :: Position -> Distance -> Position
```

The "data" Keyword

Use **data** to create a new type


```
type Position = (Int,Int)
data Direction = North | South | East | West

move :: Position -> Direction -> Position
move (x,y) West = (x-1,y)
move (x,y) East = (x+1,y)
move (x,y) South = (x,y-1)
move (x,y) North = (x,y+1)
```

we can immediately do **pattern matching** on the new type constructors

Haskell's record types

Records

```
data Person = Person { fName :: String
                     , lName :: String
                     , age :: Int
                     }
```



```
mike = Person{fName="Michael",lName="Diamond",age=43}
```



```
adam = Person{fName="Adam",lName="Horovitz",age=41}
```

```
Prelude> :type fName
fName :: Person -> String
Prelude> fName mike
“Michael”
Prelude> age mike
43
```

Basic problem: position vs name

```
type FName = String
type LName = String
type Age = Int
data PersonCurried = PersonC FName LName Age
  (deriving Show)
data PersonUncurried = PersonU (FName,LName,Age)
  (deriving Show)
```



```
data PersonRecord = PersonR { fName :: FName
                            , lName ::  LName
                            , age :: Int
                            } (deriving Show)
```

```
Prelude> :type (PersonC, PersonU, PersonR)
(PersonC, PersonU, PersonR)
  :: (FName -> LName -> Age -> PersonCurried,
      (FName, LName, Age) -> PersonUncurried,
      FName -> LName -> Int -> PersonRecord)
```

Creating values of record type

```
data PersonRecord = PersonR { fName :: FName
                            , lName ::  LName
                            , age :: Int
                            } (deriving Show)
```



```
mike = Person{fName="Michael",lName="Diamond",age=43}
adam = Person “Adam" “Horovitz" 41
mikeOlder = mike{age=44}
```

```
*Main> mike
PersonR {fName = "Michael", lName = "Diamond", age = 43}
*Main> adam
PersonR {fName = "Adam", lName = "Horovitz", age = 41}
*Main> mikeOlder
PersonR {fName = "Michael", lName = "Diamond", age = 44}
```

Accessing Fields

```
data PersonRecord = PersonR { fName :: FName
                            , lName ::  LName
                            , age :: Int
                            } (deriving Show)
```



```
mike = Person{fName="Michael",lName="Diamond",age=43}
adam = Person “Adam" “Horovitz" 41
mikeOlder = mike{age=44}
```

```
*Main> age(mike)
43
*Main> :t age
age :: PersonRecord -> Int
```

Defining Functions

```
data PersonRecord = PersonR { fName :: FName
                            , lName ::  LName
                            , age :: Int
                            } (deriving Show)
```



```
isOld (PersonR {age=a}) = a > 50
isOld’ (Person _ _ a) = a > 50
isOld’’ p = age(p) > 50
```



```
*Main> isOld(mike)
False
*Main> :t isOld
isOld :: PersonRecord -> Bool
```

More records

```
data PersonRecord = PersonR { fName :: FName
                            , lName ::  LName
                            , age :: Int
                            } (deriving Show)
```


- You can't have another record using the same field name

- If you did, you would have two access functions with the same name and different types.

Type inference and polymorphism

inference v checking

- type inference - the language works out what types things have

- type checking - the language checks that the types you have given are consistent

Basic rules of Haskell type inference

1. An identifier should be assigned the same type throughout its scope.
2. In an "if-then-else" expression, the condition must have type Bool and the "then" and "else" portions must have the same type. The type of the expression is the type of the "then" and "else" portions.
3. A user-defined function has type a → b, where a is the type of the function's parameter and b is the type of its result.
4. In a function application of the form f x, there must be types a and b such that f has type a → b and x has type a, and the application itself has type b.

Example

```
func baseAmt str = replicate
      rptAmt newStr
  where
    rptAmt = if baseAmt > 5
      then baseAmt
      else baseAmt + 2
    newStr = "Hello " ++ str
```



```
   newStr = "Hello " ++ str
(++) :: [a] -> [a] -> [a]
"Hello " :: [Char]
So a=Char and:
 str :: [Char]
 newstr :: [Char]
```

Example

```
func baseAmt str = replicate
      rptAmt newStr
   where
     rptAmt = if baseAmt > 5
       then baseAmt
       else baseAmt + 2
     newStr = “Hello “ ++ str
```

```
   5 has Num type, as does 2
(+)::Num a => a -> a -> a
So baseAmt has Num type, as
does baseAmt + 2.
(>)::Ord a => a -> a -> Bool
So baseAmt also has ord type.
And rptAmt has the same type
as baseAmt.
```

Example

```
func baseAmt str = replicate
    rptAmt newStr
  where
    rptAmt = if baseAmt > 5
      then baseAmt
      else baseAmt + 2
    newStr = “Hello “ ++ str
```



```
replicate :: Int -> a -> [a]
So baseAmt :: Int (which is
both a Num type and an Ord
type).
We know newStr :: [Char]
So a = [Char]
Hence result is [[Char]]
```

```
func :: Int -> [Char] -> [[Char]]
```

Type deduction

- When we do this we sometimes get a type like this.

- We sometimes get a contradiction (the code is untypable).
- We sometimes get some type variables left (the code is polymorphic).

Example


```
foldr :: f -> b -> c -> b
```



```
foldr f b [] = b
foldr f b (a:as) =
  f a (foldr b as)
```

Example


```
foldr :: f -> b -> [c] -> b
```



```
foldr f b [] = b
foldr f b (a:as) =
  f a (foldr f b as)
```



```
(a:as)::[c], and
(:) :: c -> [c] -> [c]
So a::c
and as :: [c]
```

Example


```
foldr f b [] = b
foldr f b (a:as) =
  f a (foldr b as)
```

```
foldr :: f -> b -> [c] -> b
```

```
(a:as)::[c], and
(:) :: c -> [c] -> [c]
So a::c
and as :: [c]
```

```
f :: c -> b -> b
```

```
So
foldr :: (c -> b -> b) -> b -> [c] -> b
```

Type classes

Type classes

defining your own type class

Haskell Type Classes

- You can think of a **type class** as a collection of types
- A **type class** defines an "interface", a list of functions that must be implemented in order to make a type an "instance" of the type class


```haskell
class Shape a where
    perimeter :: a -> Double
    area :: a -> Double
```

Here we are defining the type class **Shape**

A type **a** can be made an instance of **Shape** if we can implement the functions **perimeter :: a -> Double** and **area :: a -> Double**

Type classes

making a type an instance of a class

```haskell
class Shape a where
    perimeter :: a -> Double
    area :: a -> Double
```

```haskell
type Side = Double
type Radius = Double

data Square = Square Side
data Circle = Circle Radius
```

```haskell
instance Shape Square where
    perimeter (Square x) = 4 * x
    area (Square x) = x^2
```

```haskell
instance Shape Circle where
    perimeter (Circle r) = 2 * r * pi
    area (Circle r) = pi * r^2
```

Two new types: **Square** and **Circle**

Square is now a member of the type class Shape

Circle can also be made an member of the type class **Shape**

```
class Shape a where
    perimeter :: a -> Double
    area :: a -> Double
```

```
type Side = Double
type Radius = Double

data Square = Square Side
data Circle = Circle Radius
```

Type class

member type

```
instance Shape Square where
    perimeter (Square x) = 4 * x
    area (Square x) = x^2
```

must be a data type not a type synonym

```
instance Shape Circle where
    perimeter (Circle r) = 2 * r * pi
    area (Circle r) = pi * r^2
```

Type classes

what they are

Type classes enable function name **overloading**

```
$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/
Loading packages ...
Prelude> :type div
div :: Integral a => a -> a -> a
Prelude> :type qsort
qsort :: Ord a => [a] -> [a]
Prelude> :type elem
elem :: Eq a => a -> [a] -> Bool
Prelude> :type show
show :: Show a => a -> String
```

You can use **div** on any type in the **Integral** class

You can use **qsort** on any type in the **Ord** class

You can use **elem** on any type in the **Eq** class

You can use **show** on any type in the **Show** class

Type Class

A collection of types that support certain overloaded operations called methods

An interface that defines some behaviour

Corresponds to interfaces in OO programming

Type Class

```
$ ghci
GHCi, version 6.12.3: http://ww
Loading packages ...
Prelude> :type div
div :: Integral a => a -> a ->
Prelude> :type qsort
qsort :: Ord a => [a] -> [a]
Prelude> :type elem
elem :: Eq a => a -> [a] -> Bool
Prelude> :type show
show :: Show a => a -> String
```

any type "a" that supports division

any type "a" has a notion of order

any type "a" that supports equality test

any type "a" that can be converted into string

Type Class

```
$ ghci
G                             ttp://www.haskell.org/ghc/
L
Prelude> let f x y = if x==y then False else True
Prelude> :type f
f :: Eq a => a -> a -> Bool
Prelude> let g x y = if x<y then 0 else 1
Prelude> :type g
g :: (Ord a, Num b) => a -> a -> b
```

any type “a” that supports equality test

any type “a” that has a notion of order

any numeric type “b”

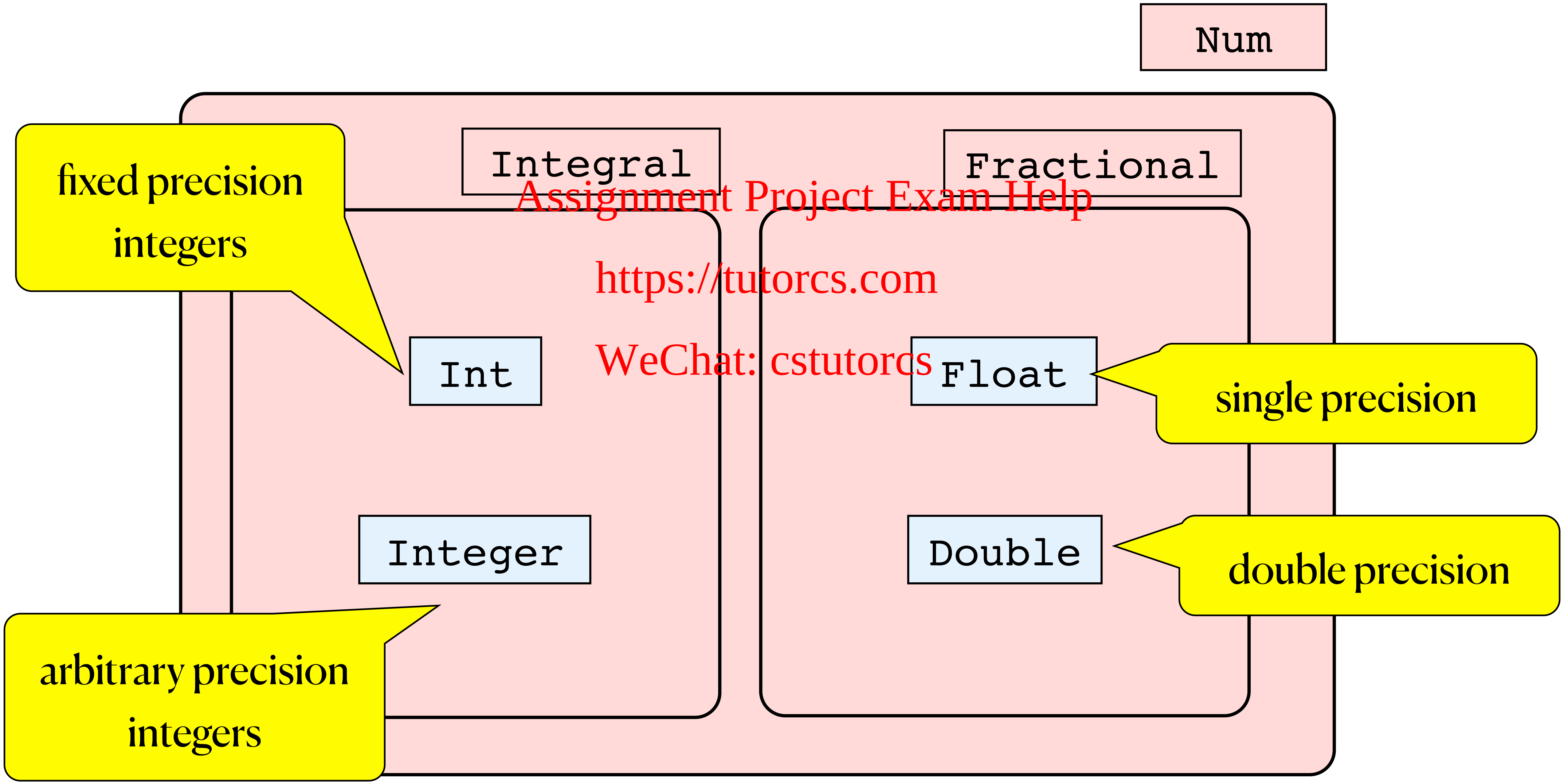
Type classes

standard built-in classes

Haskell Type Classes

- Several type classes are pre-defined in Haskell:
  - **Eq**: For types that support equality test ==
  - **Ord**: For **Eq** types that can also be ordered
  - **Show**: For types that can be mapped to **String**
  - **Read**: For types that can be mapped from **String**
  - **Num**: For numeric types, i.e. supporting +, *,...
  - **Fractional**: For numeric types that also support /

Numerical Types/Classes



Numerical Types

```
-- addition has type
(+) :: Num a => a -> a -> a
```


Fractional Types

```
-- division has type
(/) :: Fractional a => a -> a -> a
```

The Eq Type Class

```
-- all types which support equality test

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

instance Eq Bool where
   False == False = True
   True == True   = True
   _ == _         = False
```

essentially an interface describing signature of available methods

any concrete instance has to implement the methods

Sub-classing

```
-- ordered types are equality types with an extra
-- order relation

class Eq a => Ord a where
   (<),(≤),(>),(≥) :: a -> a -> Bool
   min, max :: a -> a -> a
   min x y | x ≤ y = x
           | otherwise = y
   max x y | x ≤ y = y
           | otherwise = x
```

Ordered Types

```
-- insert new element on sorted list
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x ≤ y = x : y : ys
                | otherwise = y : insert x ys
```



```
-- quick sort
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
              where
                  smaller = [a | a <- xs, a ≤ x]
                  larger = [b | b <- xs, b > x]
```

Built-in Type Classes (Show and Read)

Show / Read

```
-- types that support conversion to String

class Show a where

   show :: a -> String


-- types that support conversion from String

class Read a where

   read :: String -> a
```

Show / Read

```
$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/
Loading packages ...
Prelude> show True
"True"
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> 1 + read "2"
3
Prelude> read "[1,2,3]"
<interactive>:7:11: Ambiguous type variable...
Prelude> read "[1,2,3]" :: [Int]
[1,2,3]
```

Derived Instances

new types can be made into instances of the built-in type classes using the **deriving** keyword


```
-- For instance, Bool is defined as
data Bool = False | True
            deriving (Eq, Ord, Show, Read)

-- the following is also ok because Float is an
-- Eq type
data Shape = Circle Float | Rect Float Float
             deriving Eq
```

Week 5 Quiz

- https://qmplus.qmul.ac.uk/mod/quiz/view.php?id=1775351
- During lab this week: Friday 29th October: 14.05 - 15.55
- See web page for coverage.
- Open book.