Typeclasses

Overloading Operators: Arithmetic

The `+` operator works for a bunch of different types.

For `Integer`:

+ Int

```
λ> 2 + 3
5
```

+ Double

Deadline for 01-trees TODAY

Will post next assign 02-while

for `Double` precision floats:

```
λ> 2.9 + 3.5
6.4
```

Overloading Comparisons

Similarly we can *compare* different types of values

```
λ> 2 == 3
False
```

== Bool

```
λ>  [2.9, 3.5] == [2.9, 3.5]
True
```

== [Double]

```
λ> ("cat", 10) < ("cat", 2)
False
```

< (String, Int)

```
λ> ("cat", 10) < ("cat", 20)
True
```

Ad-Hoc Overloading

Seems unremarkable?

Languages since the dawn of time have supported "operator overloading"

- To support this kind of **ad–hoc polymorphism**
- Ad-hoc: "created or done for a particular purpose as necessary."

You really **need** to *add* and *compare* values of *multiple* types!

+, -, <, ==

Haskell has no caste system

No distinction between **operators** and **functions**

- All are first class citizens!

But then, what type do we give to *functions* like `+` and `==` ?

QUIZ

Which of the following would be appropriate types for `(+)` ?

(A) `(+) :: Integer -> Integer -> Integer`

(B) `(+) :: Double -> Double -> Double`

(C) `(+) :: a -> a -> a`

(D) *All* of the above

(E) *None* of the above

`Integer -> Integer -> Integer` is bad because?

- Then we cannot add `Double` s!

`Double -> Double -> Double` is bad because?

- Then we cannot add `Double` s!

`a -> a -> a` is bad because?

- That doesn’t make sense, e.g. to add two `Bool` or two `[Int]` or two functions!

Type Classes for Ad Hoc Polymorphism

Haskell solves this problem with **typeclasses**

- Introduced by Wadler and Blott (http://portal.acm.org/citation.cfm?id=75283)

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

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

BTW: The paper is one of the clearest examples of academic writing I have seen. The next time you hear a curmudgeon say all the best CS was done in the 60s or 70s just point them to the above.

Qualified Types

To see the right type, lets ask:

```
λ> :type (+)
(+) :: (Num a) => a -> a -> a
```

We call the above a **qualified type**. Read it as `+`

- takes in two `a` values and returns an `a` value

for any type `a` that

- *is a* `Num` or
- *implements* the `Num` interface or
- *is an instance of* a `Num`.

The name `Num` can be thought of as a *predicate* or *constraint* over types.

Some types are `Num`*s*

Examples include `Integer` , `Double` etc

- Any such values of those types can be passed to `+`.

Other types are not `Num` *s*

Examples include `Char` , `String` , functions etc,

- Values of those types *cannot* be passed to `+`.

```
λ> True + False

<interactive>:15:6:
    No instance for (Num Bool) arising from a use of ‘+’
    In the expression: True + False
    In an equation for ‘it’: it = True + False
```

Aha! *Now* those `no instance for` error messages should make sense!

- Haskell is complaining that `True` and `False` are of type `Bool`
- and that `Bool` is *not* an instance of `Num`.

Type Class is a Set of Operations

A typeclass is a collection of operations (functions) that must exist for the underlying type.

- Similar but different to Java interfaces (https://www.parsonsmatt.org/2017/01/07/how_do_type_classes_differ_from_interfaces.html)

The `Eq` Type Class

The simplest typeclass is perhaps, `Eq`

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

A type `a` is *an instance of* `Eq` if there are two functions

- `==` and `/=`

That determine if two `a` values are respectively *equal* or *disequal*.

The `Show` *Type Class*

The typeclass `Show` requires that instances be convertible to `String` (which can then be printed out)

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

Indeed, we can test this on different (built-in) types

```
λ> show 2
"2"

λ> show 3.14
"3.14"

λ> show (1, "two", ([],[],[]))
"(1,\"two\",([],[],[]))"
```

(Hey, whats up with the funny `\"` ?)

Unshowable Types

When we type an expression into `ghci`,

- it computes the value,
- then calls `show` on the result.

Thus, if we create a *new* type by

```
data Unshowable = A | B | C
```

and then create values of the type,

```
λ> let x = A
λ> :type x
x :: Unshowable
```

but then we **cannot view** them

```
λ> x

<interactive>:1:0:
    No instance for (Show Unshowable)
      arising from a use of `print' at <interactive>:1:0
    Possible fix: add an instance declaration for (Show Unshowable)
    In a stmt of a 'do' expression: print it
```

and we **cannot compare** them!

```
λ> x == x

<interactive>:1:0:
    No instance for (Eq Unshowable)
      arising from a use of `==' at <interactive>:1:0-5
    Possible fix: add an instance declaration for (Eq Unshowable)
    In the expression: x == x
    In the definition of `it': it = x == x
```

Again, the previously incomprehensible type error message should make sense to you.

Creating Instances

Tell Haskell how to show or compare values of type `Unshowable`

By **creating instances** of `Eq` and `Show` for that type:

```
instance Eq Unshowable where
  (==) A A = True           -- True if both inputs are A
  (==) B B = True           -- ...or B
  (==) C C = True           -- .. or C
  (==) _ _ = False          -- otherwise

  (/=) x y = not (x == y)   -- Test if `x == y` and negate result!
```

EXERCISE

Lets *create* an **`instance`** for `Show Unshowable`

When you are done we should get the following behavior

```
>>> x = [A, B, C]
[A, B, C]
```

Automatic Derivation

We *should* be able to compare and view `Unshowble` **automatically"**

Haskell lets us *automatically derive* implementations for some standard classes

```
data Showable = A' | B' | C'
  deriving (Eq, Show) -- tells Haskell to automatically generate instances
```

Now we have

```
λ> let x' = A'

λ> :type x'
x' :: Showable

λ> x'
A'

λ> x' == x'
True

λ> x' == B'
False
```

The `Num` *typeclass*

Let us now peruse the definition of the `Num` typeclass.

```
λ> :info Num
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

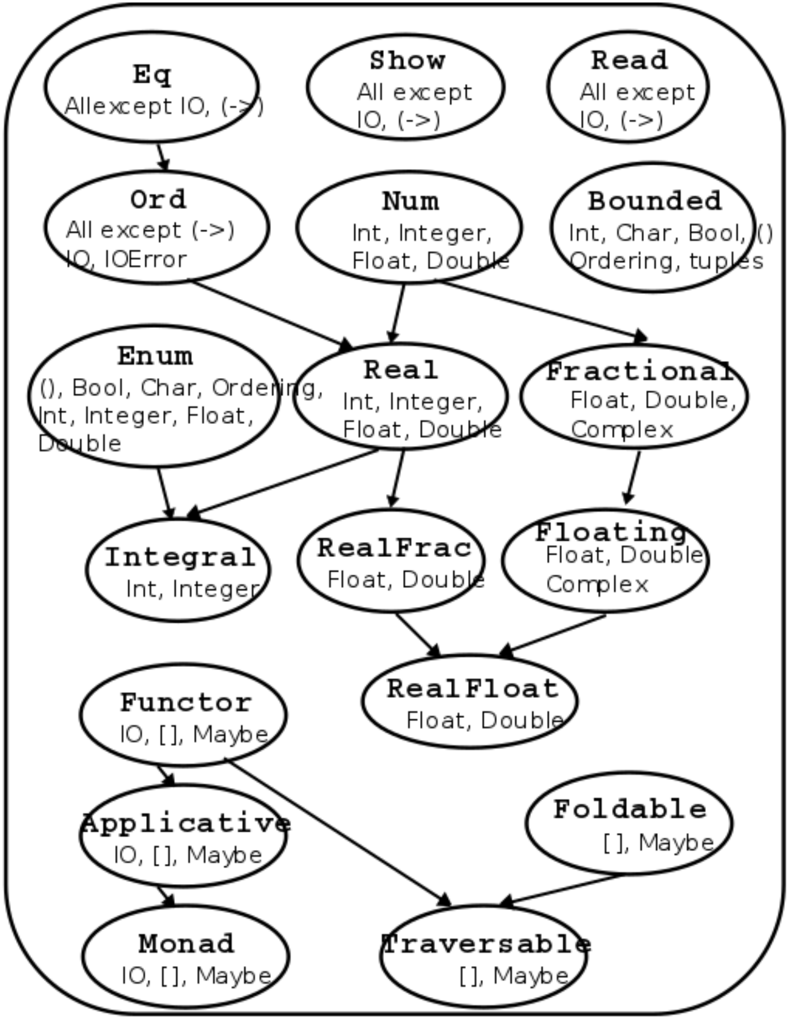
A type `a` *is an instance of* (i.e. *implements*) `Num` if

1. The type is *also* an instance of `Eq` and `Show` , and
2. There are functions to add, multiply, etc. values **of that type**.

That is, we can do *comparisons* and *arithmetic* on the values.

Standard Typeclass Hierarchy

Haskell comes equipped with a rich set of built-in classes.



Standard Typeclass Hierarchy

In the above picture, there is an edge from `Eq` and `Show` to `Num` because for something to be a `Num` it must also be an `Eq` and `Show`.

The `Ord` Typeclass

Another typeclass you've used already is the one for `Ord`ering values:

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

For example:

```
λ> 2 < 3
True

λ> "cat" < "dog"
True
```

QUIZ

Recall the datatype:

```
data Showable = A' | B' | C' deriving (Eq, Show)
```

What is the result of:

```
λ> A' < B'
```

ORD

(A) `True` **(B)** `False` **(C)** Type error **(D)** Run-time exception

Using Typeclasses

Typeclasses integrate with the rest of Haskell's type system.

Lets build a small library for *Environments* mapping keys `k` to values `v`

```
data Table k v
  = Def  v                  -- default value `v` to be used for "missing" keys
  | Bind k v (Table k v)  -- bind key `k` to the value `v`
  deriving (Show)
```

QUIZ

What is the type of `keys`

Table k v → [k]

```
keys (Def _)          = []
keys (Bind k _ rest) = k : keys rest
```

A. `Table k v -> k`

B. `Table k v -> [k]`

×**C.** `Table k v -> [(k, v)]`

Table k v → [(k,v)]

D. `Table k v -> [v]`

E. `Table k v -> v`

An API for `Table`

Lets write a small API for `Table`

```
-- >>> let env0 = set "cat" 10.0 (set "dog" 20.0 (Def 0))

-- >>> set "cat" env0
-- 10

-- >>> get "dog" env0
-- 20

-- >>> get "horse" env0
-- 0
```

Ok, lets implement!

```haskell
-- | 'add key val env' returns a new env that additionally maps `key` to `val`
set :: k -> v -> Table k v -> Table k v
set key val env = ???

-- | 'get key env' returns the value of `key` and the "default" if no value is found
get :: k -> Table k v -> v
get key env = ???
```

Oops, y u no check?

Constraint Propagation

Lets *delete* the types of `set` and `get`

- to see what Haskell says their types are!

```
λ> :type get
get :: (Eq k) => k -> v -> Table k v -> Table k v
```

We can use *any* `k` value as a *key* – if `k` is an instance of i.e. "implements" the `Eq` typeclass.

How, did GHC figure this out?

- If you look at the code for `get` you'll see that we check if two keys *are equal*!

HOMEWORK

Write an optimized version of

- `set` that ensures the keys are in *increasing* order,
- `get` that gives up and returns the "default" the moment we see a key thats larger than the one we're looking for.

(How) do you need to change the type of `Table` ?

(How) do you need to change the types of `get` and `set` ?