**Typeclasses**

**Overloading Operators: Arithmetic**

The + operator works for a bunch of different types.

For Integer:

\[
\lambda > 2 + 3
\]

5

for Double precision floats:

\[
\lambda > 2.9 + 3.5
\]

6.4

**Deadline**

**for O1-trees**

**TODAY**

Will post next assign 02-while
Overloading Comparisons

Similarly we can compare different types of values

\[
\lambda: \quad 2 \equiv 3 \quad \equiv \quad \text{Bool}
\]
False

\[
\lambda: \quad [2.9, 3.5] \equiv [2.9, 3.5] \quad \equiv \quad \text{Double}
\]
True

\[
\lambda: \quad \text{"cat", 10} < \text{"cat", 2} \quad \equiv \quad \text{String, Int}
\]
False

\[
\lambda: \quad \text{"cat", 10} < \text{"cat", 20} \quad \equiv \quad \text{String, Int}
\]
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!

\[ +, -, <, \geq \]
**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, let's ask:

\[ \lambda \text{type } (+) \]

\[ (+) :: (\text{Num } a) \Rightarrow a \Rightarrow a \Rightarrow a \]

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

• \emph{is a} \texttt{Num} \emph{or}
  • \emph{implements} the \texttt{Num} interface \emph{or}
  • \emph{is an instance of} a \texttt{Num}.

The name \texttt{Num} can be thought of as a \emph{predicate} or \emph{constraint} over types.

\textbf{Some types are} \texttt{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 \textbf{instance} for (Num Bool) arising from a use of `+'
    In the expression: True + False
    In an equation for `it': it = True + False

\textbf{Aha! Now} those no \textbf{instance} for error messages should make sense!

- Haskell is complaining that True and False are of type Bool
- and that Bool is \textit{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

```haskell
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)

```haskell
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, what's 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

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

and then create values of the type,

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

but then we **cannot view** them

```haskell
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!
\[
\lambda \ x \ == \ x
\]

<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:

```haskell
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

```haskell
>>> x = [A, B, C]
[A, B, C]
```
**Automatic Derivation**

We _should_ be able to compare and view `Unshowable automatically`.

Haskell lets us _automatically derive_ implementations for some standard classes

```haskell
  data Showable = A' | B' | C'

  deriving (Eq, Show) -- tells Haskell to automatically generate instances
```

Now we have
\[
\lambda \text{> let } x' = A'
\]

\[
\lambda \text{> :type } x'
\]
\[
x' :: \text{Showable}
\]

\[
\lambda \text{> } x'
\]
\[
A'
\]

\[
\lambda \text{> } x' == x'
\]
\[
\text{True}
\]

\[
\lambda \text{> } x' == B'
\]
\[
\text{False}
\]
The **Num** typeclass

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

\[
\lambda> \text{info Num} \\
\text{class} \ (\text{Eq a, Show a}) \Rightarrow \text{Num a where} \\
\qquad (+) :: a \to a \to a \\
\qquad (*) :: a \to a \to a \\
\qquad (-) :: a \to a \to a \\
\qquad \text{negate} :: a \to a \\
\qquad \text{abs} :: a \to a \\
\qquad \text{signum} :: a \to a \\
\qquad \text{fromInteger} :: \text{Integer} \to a
\]

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:
\lambda> :info (<)  
\textbf{class} Eq a => Ord a \textbf{where}
  ...
  (<) :: a -> a -> \texttt{Bool}
  ...

For example:

\lambda> 2 < 3
True

\lambda> "cat" < "dog"
True
QUIZ

Recall the datatype:

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

What is the result of:

```haskell
λ > A' < B'
```

(A) True (B) False (C) Type error (D) Run-time exception
Using Typeclasses

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

Let's build a small library for Environments mapping keys \( k \) to values \( v \)

```haskell
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?

\[ \text{keys (Def \_)} = [] \]
\[ \text{keys (Bind k \_ rest) = k : keys rest} \]

A. Table k v -> k

B. Table k v -> [k]

\[ \times \text{C. Table k v -> [(k, v)]} \]

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

```plaintext
-- >>> 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!
```
-- | '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

Let's delete the types of set and get

- to see what Haskell says their types are!

\[
\lambda > \texttt{type get} \\
get :: (\texttt{Eq } k) \Rightarrow k \rightarrow v \rightarrow \texttt{Table } k v \rightarrow \texttt{Table } k v
\]

We can use \texttt{any } k value as a \texttt{key} – if \texttt{k} is an instance of \texttt{Eq} typeclass.

How, did GHC figure this out?

- If you look at the code for \texttt{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 that is 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`?*