**Typeclasses**

**Announcements**

- HW 04-NANO – deadline ==> May 31, 23:59:59

```haskell
let fac = – env0 -> if n <= 0 then 1 else 1 * fac (n-1) in fac 5

let f = e1 in e2 ...

– Somehow “hack the frozen env” so that the name f is “available” in the closure’s frozen env that e1 evaluates to

VCl0s env0 “n” <if n <=0 ... >

eval (("n", 5):env0) <body>

PROBLEM: ‘fac’ is unbound in env0 and hence in ("n", 5) : env0

**Past two Weeks**

How to *implement* language constructs?

- Local variables and scope
Next two Weeks

Modern language features for structuring programs

- Type classes
- Monads

Overloading Operators: Arithmetic

The + operator works for a bunch of different types.

For Integer:

\[
\lambda > 2 + 3 \\
5
\]

for Double precision floats:

\[
\lambda > 2.9 + 3.5 \\
6.4
\]
Overloading Comparisons

Similarly we can compare different types of values

\[
\lambda \geq 2 == 3
\]
False

\[
\lambda \geq [2.9, 3.5] == [2.9, 3.5]
\]
True

\[
\lambda \geq ("cat", 10) < ("cat", 2)
\]
False

\[
\lambda \geq ("cat", 10) < ("cat", 20)
\]
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 an *insanely slick* mechanism called 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, 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

- 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 \textbf{Nums}

Examples include Integer, Double etc
- Any such values of those types can be passed to \(+\).

**Other types are not** \texttt{Num}s

Examples include \texttt{Char}, \texttt{String}, functions etc,

- Values of those types \emph{cannot} be passed to \(+\).

\lambda > \texttt{True + False}

\begin{verbatim}
<interactive>:15:6:
    No \texttt{instance} for (\texttt{Num} \texttt{Bool}) arising from a use of ‘+’
    In the expression: \texttt{True + False}
    In an equation for ‘it’: \texttt{it = True + False}
\end{verbatim}

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

- Haskell is complaining that \texttt{True} and \texttt{False} are of type \texttt{Bool}
- and that \texttt{Bool} is \texttt{not} an instance of \texttt{Num}.

**Type Class is a Set of Operations**

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

When we type an expression into ghci, it computes the value and 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
\lambda> x

<interactive>:1:0:
   No \textbf{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 \textbf{cannot compare} them!

\lambda> x == x

<interactive>:1:0:
   No \textbf{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.

\textbf{Creating Instances}

Tell Haskell how to show or compare values of type \texttt{Unshowable}.

By \textbf{creating instances} of \texttt{Eq} and \texttt{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 Let's create an instance for Show Unshowable

**Automatic Derivation**

This is silly: we *should* be able to compare and view Unshowable “automatically”!

Haskell lets us *automatically derive* functions for some classes in the standard library.

To do so, we simply dress up the data type definition with

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

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

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

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

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

\[
\lambda \text{> x' == B'}
\]
\[
\text{False}
\]

**Standard Typeclass Hierarchy**

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) \( \text{Num} \) if

1. The type is also an instance of \( \text{Eq} \) and \( \text{Show} \), and
2. There are functions for adding, multiplying, subtracting, negating etc values of that type.

In other words in addition to the “arithmetic” operations, we can compare two \( \text{Num} \) values and we can view them (as a String.)

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:

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

For example:

```haskell
λ> 2 < 3
True

λ> "cat" < "dog"
True
```

**QUIZ**

Recall the datatype:

```haskell
data Showable = A' | B' | C' deriving (Eq, Show)
```
What is the result of:

\[ \lambda A' \, B' \]

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

**An API for \( \text{Env} \)**

Let’s write a small API for \( \text{Env} \)
-- >>> let env₀ = add "cat" 10.₀ (add "dog" 20.₀ (Def 0))

-- >>> get "cat" env₀
-- 10

-- >>> get "dog" env₀
-- 20

-- >>> get "horse" env₀
-- 0

Ok, let's implement!

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

-- | 'get key env' returns the value of `key` and the "default" if no value is found
get :: k -> Env k v -> v
get key env = ???
Oops, y u no check?

**Constraint Propagation**

Let's *delete* the types of `add` and `get` and see what Haskell says their types are!

\[
\lambda > \textbf{:type} \textit{get} \\
\textit{get} :: (\textit{Eq } k) => k \to \textit{v} \to \textit{Env } k \textit{ v} \to \textit{Env } k \textit{ v}
\]

Haskell tells us that we can use any `k` value as a *key* as long as the value is an instance of 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*!
Write an optimized version of

- `add` that ensures the keys are in *increasing* order,
- `get` that gives up and returns the “default” the moment we see a key that's larger than the one we’re looking for.

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

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

---

**Explicit Signatures**

While Haskell is pretty good about inferring types in general, there are cases when the use of type classes requires explicit annotations (which change the behavior of the code.)

For example, `Read` is a built-in typeclass, where any instance `a` of `Read` has a function

```haskell
read :: (Read a) => String -> a
```

which can parse a string and turn it into an `a`. 

That is, \texttt{Read} is the \textit{opposite} of \texttt{Show}.

\textbf{Quiz}

What does the expression \texttt{read "2"} evaluate to?

(A) compile time error

(B) "2" :: String

(C) 2 :: Integer

(D) 2.0 :: Double

(E) run-time exception

Haskell is foxed!

- Doesn’t know \textit{what type} to convert the string to!
- Doesn’t know \textit{which} of the \texttt{read} functions to run!

Did we want an \texttt{Int} or a \texttt{Double} or maybe something else altogether?
Thus, here an explicit type annotation is needed to tell Haskell what to convert the string to:

```
λ> (read "2") :: Int
2

λ> (read "2") :: Float
2.0
```

Note the different results due to the different types.

**Creating Typeclasses**

Typeclasses are useful for many different things.

We will see some of those over the next few lectures.

Lets conclude today’s class with a quick example that provides a small taste.

**JSON**

*JavaScript Object Notation* or JSON (http://www.json.org/) is a simple format for transferring data around. Here is an example:
In brief, each JSON object is either

- a base value like a string, a number or a boolean,

- an (ordered) array of objects, or

- a set of string-object pairs.

**A JSON Datatype**

We can represent (a subset of) JSON values with the Haskell datatype

```haskell
{ "name" : "Ranjit",
  "age" : 41.0,
  "likes" : ["guacamole", "coffee", "bacon"],
  "hates" : [ "waiting", "grapefruit" ],
  "lunches" : [ {"day" : "monday", "loc" : "zanzibar"},
                {"day" : "tuesday", "loc" : "farmers market"},
                {"day" : "wednesday", "loc" : "harekrishna"},
                {"day" : "thursday", "loc" : "faculty club"},
                {"day" : "friday", "loc" : "coffee cart"} ]
}
data JVal
    = JStr  String
    | JNum  Double
    | JBool Bool
    | JObj  [(String, JVal)]
    | JArr  [JVal]
deriving (Eq, Ord, Show)

Thus, the above JSON value would be represented by the JVal
Serializing Haskell Values to JSON

Let's write a small library to serialize Haskell values as JSON.

We could write a bunch of functions like
doubleToJSON :: Double -> JVal
doubleToJSON = JNum

stringToJSON :: String -> JVal
stringToJSON = JStr

boolToJSON :: Bool -> JVal
boolToJSON = JBool

**Serializing Collections**

But what about collections, namely *lists* of things?

doublesToJSON :: [Double] -> JVal
doublesToJSON xs = JArr (map doubleToJSON xs)

boolsToJSON :: [Bool] -> JVal
boolsToJSON xs = JArr (map boolToJSON xs)

stringsToJSON :: [String] -> JVal
stringsToJSON xs = JArr (map stringToJSON xs)

This is getting rather tedious

- We are rewriting the same code :(
Serializing Collections (refactored with HOFs)

You could abstract by making the individual-element-converter a parameter

```haskell
xsToJSON :: (a -> JVal) -> [a] -> JVal
xsToJSON f xs = JArr (map f xs)
```

```haskell
xysToJSON :: (a -> JVal) -> [(String, a)] -> JVal
xysToJSON f kvs = JObj [ (k, f v) | (k, v) <- kvs ]
```

But this is *still rather tedious* as you have to pass in the individual data converter (yuck)
\[ \lambda > \text{doubleToJSON 4} \]
\[ \text{JNum 4.0} \]

\[ \lambda > \text{xsToJSON stringToJSON ["coffee", "guacamole", "bacon"]} \]
\[ \text{JArr [JStr "coffee", JStr "guacamole", JStr "bacon"]} \]

\[ \lambda > \text{xysToJSON stringToJSON [("day", "monday"), ("loc", "zanzibar")]} \]
\[ \text{JObj [("day", JStr "monday"), ("loc", JStr "zanzibar")]} \]

This gets more hideous when you have richer objects like

```haskell
lunches = [ [("day", "monday"), ("loc", "zanzibar")]
    , [("day", "tuesday"), ("loc", "farmers market")]
]
```

because we have to go through gymnastics like

\[ \lambda > \text{xsToJSON (xysToJSON stringToJSON) lunches} \]
\[ \text{JArr [ JObj [("day", JStr "monday"), ("loc", JStr "zanzibar")]
    , JObj [("day", JStr "tuesday"), ("loc", JStr "farmers market")]
    ]} \]

Yikes. So much for \textit{readability}

Is it too much to ask for a magical \texttt{toJSON} that \textit{just works}?
**Typeclasses To The Rescue**

Let's define a typeclass that describes types `a` that can be converted to JSON.

```haskell
class JSON a where
    toJSON :: a -> JVal
```

Now, just make all the above instances of `JSON` like so

```haskell
instance JSON Double where
    toJSON = JNum

instance JSON Bool where
    toJSON = JBool

instance JSON String where
    toJSON = JStr
```

This lets us uniformly write
\[ \lambda > \text{toJSON 4} \\
\text{JNum 4.0} \]

\[ \lambda > \text{toJSON True} \\
\text{JBool True} \]

\[ \lambda > \text{toJSON "guacamole"} \\
\text{JStr "guacamole"} \]

**Bootstrapping Instances**

The real fun begins when we get Haskell to automatically bootstrap the above functions to work for lists and key-value lists!

\[
\text{instance JSON a => JSON [a] where} \quad \\
\text{toJSON xs = JArr [toJSON x | x <- xs]} 
\]

The above says, if \( a \) is an instance of \( \text{JSON} \), that is, if you can convert \( a \) to \( \text{JVal} \) then here’s a generic recipe to convert lists of \( a \) values!

\[ \lambda > \text{toJSON [True, False, True]} \\
\text{JArr [JBln True, JBln False, JBln True]} \]

\[ \lambda > \text{toJSON ["cat", "dog", "Mouse"]} \\
\text{JArr [JStr "cat", JStr "dog", JStr "Mouse"]} \]
or even lists-of-lists!

\[
\lambda \text{ toJSON } \text{ [["cat", "dog"], ["mouse", "rabbit"]]} \\
\text{ JArr } [\text{ JArr } [\text{ JStr "cat"}, \text{ JStr "dog"}], \text{ JArr } [\text{ JStr "mouse"}, \text{ JStr "rabbit"]}]
\]

We can pull the same trick with key-value lists

\[
\text{ instance (JSON a)} \Rightarrow \text{ JSON } [(\text{ String, a})] \text{ where} \\
\text{ toJSON } \text{ kvs } = \text{ JObj } [ (k, \text{ toJSON } v) \mid (k, v) <- \text{ kvs } ]
\]

after which, we are all set!

\[
\lambda \text{ toJSON } \text{ lunches} \\
\text{ JArr } [\text{ JObj } [ (\text{ "day"}, \text{ JStr "monday"}),(\text{ "loc"}, \text{ JStr "zanzibar"})], \\
\text{ JObj } [(\text{ "day"}, \text{ JStr "tuesday"}), (\text{ "loc"}, \text{ JStr "farmers market"})]
\]

It is also useful to bootstrap the serialization for tuples (upto some fixed size) so we can easily write “non-uniform” JSON objects where keys are bound to values with different shapes.
instance (JSON a, JSON b) => JSON ((String, a), (String, b)) where
  toJSON ((k1, v1), (k2, v2)) =
    JObject [(k1, toJSON v1), (k2, toJSON v2)]

instance (JSON a, JSON b, JSON c) => JSON ((String, a), (String, b), (String, c)) where
  toJSON ((k1, v1), (k2, v2), (k3, v3)) =
    JObject [(k1, toJSON v1), (k2, toJSON v2), (k3, toJSON v3)]

instance (JSON a, JSON b, JSON c, JSON d) => JSON ((String, a), (String, b), (String, c), (String, d)) where
  toJSON ((k1, v1), (k2, v2), (k3, v3), (k4, v4)) =
    JObject [(k1, toJSON v1), (k2, toJSON v2), (k3, toJSON v3), (k4, toJSON v4)]

instance (JSON a, JSON b, JSON c, JSON d, JSON e) => JSON ((String, a), (String, b), (String, c), (String, d), (String, e)) where
  toJSON ((k1, v1), (k2, v2), (k3, v3), (k4, v4), (k5, v5)) =
    JObject [(k1, toJSON v1), (k2, toJSON v2), (k3, toJSON v3), (k4, toJSON v4), (k5, toJSON v5)]

Now, we can simply write
hs = ("name" , "Ranjit")
 ,("age" , 41.0)
 ,("likes" , ["guacamole", "coffee", "bacon"])
 ,("hates" , ["waiting", "grapefruit"])
 ,("lunches", lunches)
 )

which is a Haskell value that describes our running JSON example, and can convert it directly like so

js2 = toJSON hs

**Serializing Environments**

To wrap everything up, let's write a routine to serialize our Env

```haskell
instance JSON (Env k v) where
    toJSON env = ???
```

and presto! our serializer just works
λ> env0
Bind "cat" 10.0 (Bind "dog" 20.0 (Def 0))

λ> toJSON env0
 JObject [
   ("cat", JNum 10.0)
   , ("dog", JNum 20.0)
   , ("def", JNum 0.0)
"

That’s it for today.

We will see much more typeclass awesomeness in the next few lectures...