The Type of Equality

We test for equality, using infix operator ==

```
GHCi> 1 == 2
False
GHCi [1,2,3] == (reverse [3,2,1])
True
```

What is the type of ==?

It compares things of the same type to give a boolean results:

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

What does Haskell think?

```
GHCi> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Ad-hoc Polymorphism

Equality is “polymorphic”

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

However it is ad-hoc:

- There has to be a specific (different) implementation of it for each type
  
  ```
  primIntEq :: Int -> Int -> Bool
  primFloatEq :: Float -> Float -> Bool
  ...
  ```

- Contrast with the (parametric) polymorphism of `length`:
  
  The same program code works for all lists, regardless of the underlying element type.

  ```
  length [] = 0
  length (x:xs) = 1 + length xs
  ```

Ad-hoc polymorphism is ubiquitous

- Ad-hoc polymorphism is very common in programming languages:

<table>
<thead>
<tr>
<th>operators</th>
<th>types</th>
</tr>
</thead>
<tbody>
<tr>
<td>= ≠ &lt;= ≥</td>
<td>$T \times T \rightarrow B$, for (almost) all types $T$</td>
</tr>
<tr>
<td>+ − * /</td>
<td>$N \times N \rightarrow N$, for numeric types $N$</td>
</tr>
</tbody>
</table>

  The use of a single symbol (+, say) to denote lots of (different but related) operators, is also often called “overloading”

  In many programming languages this overloading is built-in

  In Haskell, it is a language feature called “type classes”, so we can “roll our own”.

Ad-Hoc Polymorphism
Defining (Type-)Classes in Haskell (Overloading)

- In order to define our own name/operator overloading, we:
  - need to specify the name/operator involved (e.g. `==`);
  - need to describe its pattern of use (e.g. `a -> a -> Bool`);
  - need an overarching “class” name for the concept (e.g. `Eq`).
- In order to use our operator with a given type (e.g. `Bool`, we:
  - need to give the implementation of `==` for that type (`Bool -> Bool`).
  - In other words, we give an instance of the type for the class.

Defining The Equality Class

- We define the class `Eq` as follows:
  ```haskell
class Eq a where
  (==) :: a -> a -> Bool
  ```
- The first line introduces `Eq` as a class characterising a type (here called `a`).
- The second line declares that a type belonging to this class must have an implementation of `==` of the type shown.
- `class` and `where` are Haskell keywords

Giving an instance of the Equality Class

- We define an instance of `Eq` for booleans as follows
  ```haskell
  instance Eq Bool where
  True == True    = True
  False == False  = True
  _ == _          = False
  ```
- (here `_` is a wildcard pattern matching anything).
- Now all we do is define instances for the other types for which equality is desired.
  - (In fact, in many cases, for equality, we simply refer to a primitive builtin function to do the comparison)
  - Most of this is already done for us as part of the Haskell Prelude.
- `instance` is a Haskell keyword

The “real” equality class

- In fact, `Eq` has a slightly more complicated definition:
  ```haskell
  class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) or (/=)
  x /= y    = not (x == y)
  x == y    = not (x /= y)
  ```
- First, an instance must also provide `=/=` (not-equal).
- Second, we give (circular) definitions of `==` and `/=` in terms of each other
  - The idea is that an instance need only define one of these
  - The other is then automatically derived.
  - However we may want to explicitly define both (for efficiency).
How Haskell handles a class name/operator (I)

- Consider the following (well-typed) expression:

\[ x == 3 \&\& y == False \]

(\(x\) has type \(\text{Int}\), and \(y\) is of type \(\text{Bool}\)).

- The compiler sees the symbol \(==\), notes it belongs to the \(\text{Eq}\) class, and then . . .
  - seeing \(x::\text{Int}\) deduces (via type inference) that the first \(==\) has type \(\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}\).
  - This is acceptable as it knows of such an instance of \(==\).
  - Generates code using that instance for that use of equality.
  - Does a similar analysis of the second \(==\) symbol, and generates boolean-equality code there.

How Haskell handles a class name/operator (II)

- Now consider the following (well-typed) expression:

\[ x == 3 \&\& y == False \mid z == \text{MyCons} \]

(here \(z\) has a user defined data type \(\text{MyType}\), with \(\text{MyCons}\) as a constructor).

- Assume we have not declared an instance of \(\text{Eq}\) for this type.

- The compiler, seeing the 3rd \(==\), looks for an instance for \(\text{MyType}\) of \(\text{Eq}\), and fails to find one.

- It generates a error message of the form:

\[ \text{No instance for (Eq MyType)} \]

arising from a use of ‘==’ at . . .

Possible fix: add an instance declaration for (Eq MyType).

- Note the helpful suggestion!

Standard (Prelude) Classes in Haskell

A wide range of classes and instances are provided as standard in Haskell.

- Relations \(\text{Eq, Ord}\)
- Enumeration \(\text{Enum, Bounded}\)
- Numeric \(\text{Num, Real, Integral, Fractional, Floating}\)
  - \(\text{RealFrac, RealFloat}\)
- Textual \(\text{Show, Read}\)
- Categorical \(\text{Functor, Monad}\)

Guided tour: the \(\text{Eq a}\) Class

Class Members

\[
\begin{align*}
(==) :: & a \rightarrow a \rightarrow \text{Bool} \\
(\neq) :: & a \rightarrow a \rightarrow \text{Bool}
\end{align*}
\]

Instances \((), \text{Bool, Char, Int, Integer, Float, Double, Ordering, IOError, Maybe, Either, [a], (a,b), (a,b,c)}\)

Comments

There is no instance of \(\text{Eq}\) for any function type.

Function equality is undecidable:

\[ f = g \iff f x = g x \] for all \(x\) in input type of \(f\) and \(g\).

We have yet to see how to define instances for type-constructors rather than types.
Guided tour: the *Ord* a Class

**Class Members**

- `compare :: a -> a -> Ordering`
- `(<=), (>=), (<), (>) :: a -> a -> Bool`
- `max, min :: a -> a -> a`

**Instances**

- `()`, `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`, `Ordering`, `Maybe`, `Either`, `[a]`, `(a,b)`, `(a,b,c)`

**Comments**

Almost everything with equality also has ordering defined.
(expect `IOError`).

---

The *Ordering* Type

- It is a straightforward `data` definition
  ```haskell
data Ordering = LT | EQ | GT```
- It represents the three possible outcomes of a order comparison.

---

Guided tour: the *Enum* a Class

**Class Members**

- `succ, pred :: a -> a`
- `toEnum :: Int -> a`
- `fromEnum :: a -> Int`
- `enumFrom :: a -> [a]`
- `enumFromThen :: a -> a -> [a]`
- `enumFromTo :: a -> a -> [a]`
- `enumFromThenTo :: a -> a -> a -> [a]`

**Instances**

- `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`

**Comments**

Basically types for which notation `[start .. end]` makes sense.
The `Float`, `Double` and `Double` instances for `Enum` should be used with care (rounding errors).

---

Guided tour: the *Bounded* a Class

**Class Members**

- `minBound :: a`
- `maxBound :: a`

**Instances**

- `()`, `Bool`, `Char`, `Int`, `Ordering`, `(a,b)`, `(a,b,c)`

**Comments**

Types that have a (natural) minimum and maximum value.
Guided tour: the **Num** a Class

Class Members

\[(+), (-), (*) \quad :: \quad a \rightarrow a \rightarrow a\]

- `negate` :: a \rightarrow a
- `abs`, `signum` :: a \rightarrow a
- `fromInteger` :: Integer \rightarrow a

Instances `Int`, `Integer`, `Float`, `Double`

Comments Basic integer-like numbers.
(Note lack of any form of division).

Guided tour: the **Real** a Class

Class Members

\[
toRational \quad :: \quad a \rightarrow \text{Rational}\]

Instances `Int`, `Integer`, `Float`, `Double`

Comments A strange class, basically those numbers that can be expressed as rationals (ratio of two integers).

The **Rational** Type

- The Rational type, defined in library module `Data.Ratio`
  
  type Rational = Ratio Integer

- The Ratio type constructor forms a pair
  
  data Ratio a = a :% a

- We can build Ratio values using infix operator \(%\).
Guided tour: the **Fractional a Class**

**Class Members**

- `(/) :: a -> a -> a`
- `recip :: a -> a`
- `fromRational :: Rational -> a`

**Instances** Float, Double

**Comments** Number types that handle real-number division.

Guided tour: the **Floating a Class**

**Class Members**

- `pi :: a`
- `exp, log, sqrt :: a -> a`
- `(**, logBase :: a -> a -> a`
- `sin, cos, tan :: a -> a`
- `asin, acos, atan :: a -> a`
- `sinh, cosh, tanh :: a -> a`
- `asinh, acosh, atanh :: a -> a`

**Instances** Float, Double

**Comments** All the well-known real functions.

Guided tour: the **RealFrac a Class**

**Class Members**

- `properFraction :: (Integral b) => a -> (b,a)`
- `truncate, round :: (Integral b) => a -> b`
- `ceiling, floor :: (Integral b) => a -> b`

**Instances** Float, Double

**Comments** Numbers supporting conversions from real to integral forms.

Guided tour: the **RealFloat a Class**

**Class Members**

- `floatRadix :: a -> Integer`
- `floatDigits :: a -> Int`
- `floatRange :: a -> (Int,Int)`
- `decodeFloat :: a -> (Integer,Int)`
- `encodeFloat :: Integer -> Int -> a`
- `exponent :: a -> Int`
- `significand :: a -> a`
- `scaleFloat :: Int -> a -> a`
- `isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE :: a -> Bool`
- `atan2 :: a -> a -> a`

**Instances** Float, Double

**Comments** Numbers supporting a floating-point representation plus some IEEE 754 support.
Guided tour: the **Show** a Class

Class Members

```haskell
showsPrec :: Int -> a -> ShowS
show    :: a -> String
showList :: [a] -> ShowS
```

Instances

- ()
- Bool
- Char
- Int
- Integer
- Float
- Double
- IOError
- Maybe
- Either
- Ordering
- [a]
- (a,b)

Comments

Ways to produce a textual display of a type. showsPrec takes an initial precedence argument for pretty-printing.

There is no instance of Show for functions.

---

Guided tour: the **Read** a Class

Class Members

```haskell
readsPrec :: Int -> ReadS a
readList :: ReadS [a]
```

Instances

- ()
- Bool
- Char
- Int
- Integer
- Float
- Double
- IOError
- Maybe
- Either
- Ordering
- [a]
- (a,b)

Comments

These are parsers that parse the output of show.

---

The **ShowS** type

- Define as:
  ```haskell
type ShowS = String -> String
```

- Using string building functions is often more efficient than directly generating strings.

---

The **ReadS** type

- Define as:
  ```haskell
type ReadS a = String -> [(a,String)]
```

- These functions take a string and return a list of successful parses.
Guided tour: the **Functor** $f$ Class

**Class Members**

```
fmap :: (a -> b) -> f a -> f b
```

**Instances** `Maybe`, `IO`, `[]`

**Comments** Basically any datatype where mapping a function makes sense.

This is in fact a type-`constructor` class.

Guided tour: the **Monad** $m$ Class

**Class Members**

```
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
fail :: String -> m a
```

**Instances** `Maybe`, `IO`, `[]`

**Comments** Types where sequencing of “actions” makes sense.

Like **Functor**, this is also a type-`constructor` class.

Instantiating Example

- Consider the following Binary Tree type
  ```
data STree = Leaf String | Br STree STree
  ```

- Displaying this, checking for equality and ordering are all sensible things to do

- We shall define instances for `STree` for classes `Eq`, `Ord`, and `Show`

**STree instance for Eq**

- The instance declaration
  ```
  instance Eq STree where
  (Leaf s1) == (Leaf s2) = s1 == s2
  (Br t11 t12) == (Br t21 t22) = t11 == t21 && t12 == t22
  _ == _ = False
  ```

- It is the obvious declaration of equality

- The definition of `/=` can be got from that for `==`
STree instance for Ord

▶ The instance declaration

```haskell
instance Ord STree where
    compare (Leaf s1) (Leaf s2) = compare s1 s2
    compare (Leaf _) (Br _ _) = LT
    compare (Br _ _) (Leaf _) = GT
    compare (Br t11 t12) (Br t21 t22)
        | t11 == t21 = compare t12 t22
        | otherwise = compare t11 t21
```

▶ We assume `Leaf` are smaller than `Br`

▶ We only need to define `compare` — the others are derivable from it.

STree instance for Show

▶ The instance declaration

```haskell
instance Show STree where
    show (Leaf s) = "(Leaf "++s++")"
    show (Br t1 t2) = "(Br "++show t1 ++" ++show t2++)"
```

▶ We display the tree here using the same Haskell syntax we use to create the values.

▶ Again, `showsPrec` and `showList` can be derived.

---

Deriving Instances

▶ For certain (standard) type-classes, we can ask Haskell to automatically generate instances for user-defined types.

▶ So the following code replaces all the instance declaration shown above

```haskell
data STree = Leaf String | Br STree STree
deriving (Eq,Ord,Show)
```

▶ `deriving` is another Haskell keyword.

▶ Deriving can be done for classes:
  * `Eq`, `Ord`, `Enum`, `Bounded`, `Show` and `Read`.

Not the Whole Story

There are some aspects of the typeclass system that haven’t been discussed yet

▶ Some classes depend on other classes

▶ Some classes are themselves polymorphic

▶ Some classes are associated with type constructors
Classes based on other Classes

▶ Here is part of the class declaration for \( \text{Ord} \):

\[
\text{class \ (Eq \ a) \Rightarrow \text{Ord} \ a \ where}\\
\text{compare} \quad :: \ a \rightarrow a \rightarrow \text{Ordering}\\
(\langle\rangle, (\langle\rangle), (\rangle\rangle), (\rangle\rangle) \quad :: \ a \rightarrow a \rightarrow \text{Bool}\\
\text{max, min} \quad :: \ a \rightarrow a \rightarrow a\\
\text{compare} \ x \ y\\
| \ x == y \quad = \text{EQ}\\
| \ x <= y \quad = \text{LT}\\
| \ otherwise \quad = \text{GT}\\
\]

▶ The notation \((\text{Eq} \ a) \Rightarrow\) is a context, stating that the \(\text{Ord}\) class depends on the \(\text{Eq}\) class (why?)

▶ In order to define \text{compare}, we have to use ==

▶ So, for a type to belong to \(\text{Ord}\), it must belong to \(\text{Eq}\)

▶ Think of it as a form of inheritance

“Polymorphic” Type Classes (I)

How might we define an \(\text{Eq}\) instance for lists?

▶ For \([\text{Bool}]\)

\[
\text{instance Eq [Bool] where}\\
[] == [] = \text{True}\\
(b1:bs1) == (b2:bs2) = b1 == b2 \&\& bs1 == bs2\\
_ == _ = \text{False}\\
\]

▶ For \([\text{Int}]\)

\[
\text{instance Eq [Int] where}\\
[] == [] = \text{True}\\
(i1:is1) == (i2:is2) = i1 == i2 \&\& is1 == is2\\
_ == _ = \text{False}\\
\]

▶ The red == above are where we use equality for \(\text{Bool}\) and \(\text{Int}\) respectively.

▶ Can’t we do this polymorphically?

“Polymorphic” Type Classes (II)

▶ We can!

\[
\text{instance (Eq a) => Eq [a] where}\\
[] == [] = \text{True}\\
(x1:xs1) == (x2:xs2) = x1 == x2 \&\& xs1 == xs2\\
_ == _ = \text{False}\\
\]

▶ We can define equality on \([a]\) provided we have equality set up for \(a\)

▶ Here we are defining equality for a type constructor \([\ ]\) for lists) applied to a type \(a\):

▶ so the class refers to a type built with a constructor
Consider the class declaration for **Functor**

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Here we are associating a class with a *type-constructor* `f`

- not with a type
- See how in the type signature `f` is applied to type variables `a` and `b`.
- So, `f` is something that takes a type as argument to produce a type.

**Type Constructor Examples**

- The **Maybe** type-constructor
  ```haskell
data Maybe a = Nothing | Just a
```

- The **IO** type-constructor
  ```haskell
data IO a = ...
```

- The **[]** type-constructor
  The type we usually write as `[a]` can be written as `[] a`
  i.e. the application of list constructor `[]` to a type `a`.

**Instances of Functor**

- **Maybe as a Functor**
  ```haskell
  instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
  ```

- **[] as a Functor**
  ```haskell
  instance Functor [] where
    fmap = map
  ```

- Both the above are straight from the Prelude.

**Class Contexts**

- We have seen notation in class declarations stating the one class depends on another, e.g.
  ```haskell
  (Eq a) => Ord a
  ```

- However, we also see such contexts in function type signatures:
  ```haskell
  sum :: (Num a) => [a] -> a
  ```

- What are they telling us about such functions?
Class Contexts in Function Types

- Consider a possible definition of `sum`:
  ```haskell
  sum [] = 0
  sum (n:ns) = n + sum ns
  ```
- The function is almost polymorphic, but for the use of `+`.
- It will work provided the element type belongs to the `Num` class.
- This is exactly what the type signature says:
  ```haskell
  sum :: (Num a) => [a] -> a
  ```
  "`sum` transforms a list of `a`s to a result of type `a`, provided type `a` is in the `Num` class (i.e. supports `+`)."

Instances and Type Declarations

- A type can only have one instance associated with any given class.
- A type synonym therefore cannot have its own instance declaration.
  ```haskell
  type MyType a = ...
  ```
  It simply is a shorthand for an existing type.
- A user-defined algebraic datatype can have instance declarations
  ```haskell
  data MyData a = ...
  ```
  In general we need to do this for `Eq`, `Show` in any case.
- A user-cloned (new) type can also have instance declarations
  ```haskell
  newtype MyNew a = ...
  ```
  A key use of `newtype` is to allow instance declarations for existing types (now "re-badged").

Designing a Name Class

- Consider datatypes for defining Predicates and Expressions
  ```haskell
  data Pred = T | F | Obs Expr | Not Pred | And Pred Pred
              deriving (Eq,Ord,Show)
  data Expr = Num Int | Var String | Rel String Expr Expr
              deriving (Eq,Ord,Show)
  ```
- Imagine we have variables called `ok` and `ok'`.
- Their names are represented as Strings: "ok", "ok'"
- As Expression Variables we need to use `(Var "ok")` and `(Var "ok'")`
- As Predicates, we need to wrap further: `(Obs (Var "ok"))` and `(Obs (Var "ok'"))`
- Wouldn't it be nice if we could use Haskell names `ok` and `ok'` to cover all of these uses?

An Ok Class

- One approach is to represent each variable by its own class
  ```haskell
  class Ok a where ok :: a
  ```
  - So a member of this class has to have a value called `ok`
- A string instance gives its name as a string
  ```haskell
  instance Ok String where ok = "ok"
  ```
- An Expr instance wraps it up as a Var
  ```haskell
  instance Ok Expr where ok = Var "ok"
  ```
- An Pred instance wraps it up as a Var inside an Obs
  ```haskell
  instance Ok Pred where ok = Obs (Var "ok")
  ```
Problems with GHCi

▶ If we load this into GHCi we get an error message:

```
Illegal instance declaration for 'Name String'
(All instance types must be of the form (T t1 ... tn)
where T is not a synonym.
Use -XTypeSynonymInstances if you want to disable this.)
```

▶ Remember, String is a synonym for [Char]

▶ We can allow this by invoking GHCi as suggested in fact we go one better, and enable a whole raft of extensions

```
ghci -fglasgow-exts
```

## Using Ok

▶ In order to use ok we need to supply enough context for Haskell to figure out which instance is met

```
GHCi> ok
<interactive>:1:0:
Ambiguous type variable ‘a’ in the constraint:
‘Ok a’ arising from a use of ‘ok’ at <interactive>:1:0-1
Probable fix:
add a type signature that fixes these type variable(s)
```

▶ In isolation, we need to specify the type:

```
GHCi> ok :: Expr
Var "ok"
```

▶ Given enough context, Haskell can figure it out:

```
GHCi> And ok (And (Obs (Rel "<" (Num 1) ok)) (Obs (Var ok)))
And (Obs (Var "ok"))
(And (Obs (Rel "<" (Num 1) (Var "ok"))) (Obs (Var "ok"))
```

### An Name Class

▶ A class per name seems excessive — why not one that has the name as a parameter?

```
class Name a where name :: String -> a
```

▶ So a member of this class has to have a name-generating function called name

▶ A string instance gives its name as a string

```
instance Name String where name x = x
```

▶ An Expr instance wraps it up as a Var

```
instance Name Expr where name x = Var x
```

▶ An Pred instance wraps it up as a Var inside an Obs

```
instance Name Pred where name x = Obs (Var x)
```

▶ We can now define variables:

```
wait = name "wait"
```

Problems with GHCi (Again!)

▶ If we load this into GHCi we get another error message:

```
Ambiguous type variable ‘a1’ in the constraint:
‘Name a1’ arising from a use of ‘name’ at l15.hs:43:6-16
Possible cause: the monomorphism restriction applied to the following:
wait :: a1 (bound at l15.hs:43:0)
Probable fix: give these definition(s) an explicit type signature
or use -XNoMonomorphismRestriction
```

▶ What is the "monomorphism restriction"?

▶ Good question

▶ a subtle (very) interaction between type-inference, classes and execution efficiency

▶ Best ignored and side-stepped

▶ We can allow this by invoking GHCi as suggested

```
ghci -fglasgow-exts -XNoMonomorphismRestriction
```
Now we can use `wait`, supplying sufficient context

```haskell
GHCi> wait :: String
"wait"
GHCi> wait :: Expr
Var "wait"
GHCi> wait :: Pred
Obs (Var "wait")
```