What is the type of (==)?
E.g. the following both work:

\[
1 == 2 \\
'a' == 'b'
\]

I.e., (==) can be used to compare both numbers and characters.

Maybe (==) :: a -> a -> Bool?

No!!! Cannot work uniformly for arbitrary types!

A function like the identity function

\[
\text{id} :: a -> a \\
\text{id} x = x
\]

is polymorphic precisely because it works uniformly for all types: there is no need to “inspect” the argument.

In contrast, to compare two “things” for equality, they very much have to be inspected, and an appropriate method of comparison needs to be used.

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.
- But to add properly, we must understand what we are adding.
- Not every type admits addition.
Haskell Overloading (4)

Idea:
- Introduce the notion of a type class: a set of types that support certain related operations.
- Constrain those operations to only work for types belonging to the corresponding class.
- Allow a type to be made an instance of (added to) a type class by providing type-specific implementations of the operations of the class.

The Type Class Eq

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

(==) is not a function, but a method of the type class Eq. It’s type signature is:

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

Eq a is a class constraint. It says that the equality method works for any type belonging to the type class Eq.

Instances of Eq (1)

Various types can be made instances of a type class like Eq by providing implementations of the class methods for the type in question:

instance Eq Int where
  x == y = primEqInt x y

instance Eq Char where
  x == y = primEqChar x y

Instances of Eq (2)

Suppose we have a data type:

data Answer = Yes | No | Unknown

We can make Answer an instance of Eq as follows:

instance Eq Answer where
  Yes == Yes = True
  No == No = True
  Unknown == Unknown = True
  _ == _ = False
Instances of `Eq` (3)

Consider:

```haskell
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Can `Tree` be made an instance of `Eq`?

Instances of `Eq` (4)

Yes, for any type `a` that is already an instance of `Eq`:

```haskell
instance (Eq a) => Eq (Tree a) where
    Leaf a1 == Leaf a2 = a1 == a2
    Node t1l t1r == Node t2l t2r = t1l == t2l && t1r == t2r
    _ == _ = False
```

Note that `==` is used at type `a` (whatever that is) when comparing `a1` and `a2`, while the use of `==` for comparing subtrees is a recursive call.

Derived Instances (1)

Instance declarations are often obvious and mechanical. Thus, for certain *built-in* classes (notably `Eq`, `Ord`, `Show`), Haskell provides a way to *automatically derive* instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

```haskell
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
deriving Eq
```

Derived Instances (2)

GHC provides some additional possibilities. With the extension `-XGeneralizedNewtypeDeriving`, a new type defined using `newtype` can "inherit" any of the instances of the representation type:

```
newtype Time = Time Int deriving Num
```

With the extension `-XStandaloneDeriving`, instances can be derived separately from a type definition (even in a separate module):

```
deriving instance Eq Time
deriving instance Eq a => Eq (Tree a)
```
**Class Hierarchy**

Type classes form a hierarchy. E.g.:

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

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.

**Haskell vs. OO Overloading (1)**

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types.

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider read:

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

Note: overloaded on the **result** type! A method that converts from a string to any other type in class Read!

**Haskell vs. OO Overloading (2)**

```haskell
> let xs = [1,2,3] :: [Int]
> let ys = [1,2,3] :: [Double]
> xs
[1,2,3]
> ys
[1.0,2.0,3.0]
> (read "42" : xs)
[42,1,2,3]
> (read "42" : ys)
[42.0,1.0,2.0,3.0]
> read "'a'" :: Char
'a'
```

**Implementation (1)**

The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.

Thus, internally `==` is a **higher order function** with three arguments:

```
(==) eqF x y = eqF x y
```
Implementation (2)

An expression like

```
1 == 2
```

is essentially translated into

```
(==) primEqInt 1 2
```

Implementation (3)

So one way of understanding a type like

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

is that Eq a corresponds to an extra implicit argument.

The implicit argument corresponds to a so called directory, or tuple/record of functions, one for each method of the type class in question.

Some Basic Haskell Classes (1)

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

class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a

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

Some Basic Haskell Classes (2)

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

Quiz: What is the type of a numeric literal like 42? What about 42.0? Why?

Haskell's numeric literals are overloaded. E.g. 42 is expanded into fromInteger 42.
A Typing Conundrum (1)

Overloaded (numeric) literals can lead to some surprises.

What is the type of the following list? Is it even well-typed???

\[ 1, [2, 3] \]

Surprisingly, it is well-typed:

```
> :type [1, [2,3]]
[1, [2,3]] :: (Num t, Num t) => [[t]]
```

Why?

---

A Typing Conundrum (2)

The list is expanded into:

```
[ fromInteger 1,
  fromInteger 2, fromInteger 3 ]
```

Thus, if there were some type \( t \) for which \([t]\) were an instance of \( \text{Num} \), the 1 would be an overloaded literal of that type, matching the type of the second element of the list.

Normally there are no such instances, so what almost certainly is a mistake will be caught. But the error message is rather confusing.

---

Application: Automatic Differentiation

- **Automatic Differentiation**: method for augmenting code so that derivative(s) computed along with main result.
- Purely algebraic method: arbitrary code can be handled
- Exact results
- But no separate, self-contained representation of the derivative.

---

Automatic Differentiation: Key Idea

Consider a code fragment:

\[
\begin{align*}
z_1 &= x + y \\
z_2 &= x \times z_1
\end{align*}
\]

Suppose the derivatives of \( x \) and \( y \) w.r.t. common variable is available in the variables \( x' \) and \( y' \).

Then code can be augmented to compute derivatives of \( z_1 \) and \( z_2 \):

\[
\begin{align*}
z_1 &= x + y \\
z_1' &= x' + y' \\
z_2 &= x \times z_1 \\
z_2' &= x' \times z_1 + x \times z_1'
\end{align*}
\]
Approaches

- Source-to-source translation
- Overloading of arithmetic operators and mathematical functions

The following variation is due to Jerzy Karczmarczuk. Infinite list of derivatives allows derivatives of arbitrary order to be computed.

### Functional Automatic Differentiation (1)

Introduce a new numeric type \( C \): value of a continuously differentiable function at a point along with all derivatives at that point:

```haskell
data C = C Double CvalC (C a _) = a
derC (C _ x') = x'
```

### Functional Automatic Differentiation (2)

Constants and the variable of differentiation:

```haskell
zeroC :: C
zeroC = C 0.0 zeroC

constC :: Double -> C
constC a = C a zeroC

dVarC :: Double -> C
dVarC a = C a (constC 1.0)
```

### Functional Automatic Differentiation (3)

Part of numerical instance:

```haskell
instance Num C where
  (C a x') + (C b y') =
    C (a + b) (x' + y')

  (C a x') - (C b y') =
    C (a - b) (x' - y')

  x@(C a x') * y@(C b y') =
    C (a * b) (x' * y + x * y')

fromInteger n =
  constC (fromInteger n)
```
**Functional Automatic Differentiation (4)**

Computation of \( y = 3t^2 + 7 \) at \( t = 2 \):

\[
\begin{align*}
t &= \text{dVarC } 2 \\
y &= 3 \times t \times t + 7
\end{align*}
\]

\[
\begin{align*}
\text{valC } y &\Rightarrow 19.0 \\
\text{valC } (\text{derC } y) &\Rightarrow 12.0 \\
\text{valC } (\text{derC } (\text{derC } y)) &\Rightarrow 6.0 \\
\text{valC } (\text{derC } (\text{derC } (\text{derC } y))) &\Rightarrow 0.0
\end{align*}
\]

**Functor (1)**

A Functor is a notion that originated in a branch of mathematics called Category Theory.

However, for our purposes, we can think of functors as type constructors \( T \) (of arity 1) for which a function \( \text{map} \) can be defined:

\[
\text{map} :: (a \rightarrow b) \rightarrow Ta \rightarrow Tb
\]

that satisfies the following laws:

\[
\begin{align*}
\text{map id} &= \text{id} \\
\text{map}(f \circ g) &= \text{map } f \circ \text{map } g
\end{align*}
\]

**Functor (2)**

Common examples of functors include (but are certainly not limited to) *container types* like lists:

\[
\begin{align*}
\text{mapList} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{mapList } _[] &= [] \\
\text{mapList } f \ (x:xs) &= f \ x : \text{mapList } f \ xs
\end{align*}
\]

and trees:

\[
\begin{align*}
\text{mapTree} :: (a \rightarrow b) \rightarrow \text{Tree } a \\
&\rightarrow \text{Tree } b
\end{align*}
\]

**Functor (3)**

Another important example is Haskell’s option type \textit{Maybe}:

\[
\begin{align*}
\text{data } \text{Maybe } a &= \text{Nothing } | \text{Just } a \\
\text{mapMaybe} :: (a \rightarrow b) \rightarrow \text{Maybe } a \\
&\rightarrow \text{Maybe } b \\
\text{mapMaybe } _\text{Nothing} &= \text{Nothing} \\
\text{mapMaybe } f \ (\text{Just } x) &= \text{Just } (f \ x)
\end{align*}
\]
Functor (4)

Of course, the notion of a functor with a type class in Haskell:

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

Instances are provided for [], Maybe, and many other types in Haskell's standard prelude.

However, Haskell's type system is not powerful enough to enforce the functor laws.

In general: the responsibility for ensuring that an instance respects any laws associated with a type class rests squarely with the programmer.

Functor (5)

Note that the type of `fmap` can be read:

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

That is, we can see `fmap` as promoting a function to work in a different context.

Reading