Type = set of "similar" objects

e.g., Int, Bool, [Int], [[Int]], Int → [Int],
(Int, [Int], Int → Int), ...

General idea: one uses a type constructor to create new types from already existing types.

- (ty_constr type₁ ... typeₙ), n ≥ 0

  type constructor, e.g., Int, Bool, Char, ...

  are pre-defined type constructors of arity 0

More type constructors can be defined by the user (see later).
Moreover, [], [..], →, (..) are pre-defined type constructors with special syntax.

- [type] : [..] is a type constructor of arity 1

  stands for the type of lists where all elements have the same type

  e.g., [Int], [[Int]], ...

- (type₁ → type₂) : → is a type constructor of arity 2

  stands for the type of functions from type₁ to type₂
e.g. \([\text{Int}] \rightarrow \text{Bool}, \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}), \ldots\)

\((\text{type}_1, \ldots, \text{type}_n), \ n \geq 0 : (\ldots)\) is a type constructor of arbitrary arity

\((\cdot)\) is a type with only one object: \((\cdot)\)

\(\text{type} \) is the same as \(\text{type} :\)

\[
\begin{array}{c}
5 \\
\text{type Int}
\end{array}
\]

\[
\begin{array}{c}
\text{type Int} \\
= (\text{Int})
\end{array}
\]

\((\text{Int}, [\text{Int}], \text{Int} \rightarrow \text{Int})\) is a type

\([5, [1,2,3]], \text{square}\) is an expression of this type

\(\text{Var}: \) a type variable is also a type.

Needed for parametric polymorphism.

**Parametric polymorphism**

**Polymorphism:** one can apply the same function to arguments of different types

**Functional languages**

*parametric polymorphism:* the same implementation of a function is used for arguments of different
tyros

- ad hoc polymorphism: function has several different implementations (with the same function name).
- Type of the arguments determines which implementation is executed.

Nowadays, many languages have both forms of polymorphism (e.g., Haskell, Java since Java 5, ...)

First: parametric polymorphism
(Slide 14)

\[ \text{id} :: a \rightarrow a \]
\[ \text{id} \ x = x \]

Type of append (`++`):

\[ (++ :: [a] \rightarrow [a] \rightarrow [a]) \]

Now a function of type `\text{type}_1 \rightarrow \text{type}_2`

- can be applied to an argument of type `\text{type}_1`
- if there is a most general substitution \( \sigma \) of type variables
- such that \( \sigma(\text{type}_1) = \sigma(\text{type}) \).
- Result has type: \( \sigma(\text{type}_2) \).
Example:

```
Ex:  [] True ++ [ ]
actual types:  [] Bool  [] b
expected types:  [] a  [] a
```

Result:  [] a

We can unify these types:

\[ \sigma(a) = \text{Bool} \quad \sigma(b) = \text{Bool} \]

Result:

\[ \sigma([a]) = [] \text{Bool} \]

```
Ex:  [] [] ++ [True]
actual types:  [] b  [] [Bool]
expected types:  [] a  [] a
```

Same solution:

\[ \sigma(a) = \sigma(b) = \text{Bool} \]

**Type definitions: introducing new types**

Goal: define new type constructors

The declaration of new types is only possible on top-level (i.e., not in local declarations).

**Slide 25:** distinguish between

- `topdecl` and `decl`
- only on top level,
i.e., a program is a sequence of top declarations

**Type synonyms** (keyword "type")

```plaintext
type Position = (Float, Float)
type String = [Char]
type Pair a b = (a, b)
```

Introduces a new type constructor of arity 2

Haskell makes no difference between `Position`, `(Float, Float)`, or `Pair Float Float`

One can also introduce completely new types by the keyword **data**.

Here, the programmer provides a context-free grammar (in EBNF-like notation) to define a new type. (Slide 16)

```plaintext
data Color = Red | Yellow | Green
```

New type constructor of arity 0, i.e., `Color` is a type

```
Red :: Color
```

```plaintext
data MyBool = MyTrue | MyFalse
```

Now one can define algorithms with these new types and
use pattern matching, as for pre-defined types (Slide 16).

To print objects on the screen, Haskell uses a function "show" to convert them to strings. The function "show" is pre-defined for many types (Int, Bool, [Int]...,) but not for user-defined types.

To implement "show" for user-defined types:
- do it manually (needs "type classes", next lecture)
- generate a default implementation of "show" automatically: add "deriving Show" at the end of the data-declaration

We now show how to use "data" declarations for types with infinitely many objects (Slide 17):

```
data Nats = Zero | Succ Nats
```

Data constructors:

```
Zero :: Nats
Succ :: Nats -> Nats
```

```
Zero \leq 0
Succ Zero \leq 1
Succ (Succ Zero) \leq 2
```

One can also introduce new type constructors of

\text{arity} \geq 0 \text{ (i.e., introduce new polymorphic types)}

\text{Ex:} introduce data type for lists (Slide 18).
\[
\text{data list } a = \text{Nil} \mid \text{Cons } a \ (\text{list } a)
\]

The newly introduced type is \(\text{list } a\)

\[
\begin{align*}
\text{Nil} & : \text{list } a \\
\text{Cons} & : a \to \text{list } a \to \text{list } a
\end{align*}
\]

\[
\text{len} : \text{list } a \to \text{Nats}
\]

\[
\text{len Nil} = \text{Zero}
\]

\[
\text{len } (\text{Cons } x \times x s) = \text{Succ } (\text{len } x s)
\]

In a data declaration

\[
\text{data } \underline{\text{by}} \text{const } \text{var}_1 \ldots \text{var}_n = \ldots
\]

the right-hand side may not contain any type variables except \(\text{var}_1, \ldots, \text{var}_n\).

**Type Classes**

- Type class = set of types
- Elements of a type class are called instances
- Functions in these instances may have the same name, but different implementations (ad-hoc polymorphism, overloading)

\[
(=\ =), (\neq) : : a \to a \to \text{Bool}
\]
2 == 3  is  False
\[[1,2] == [1,2]  is  True
True == False  is  False

But:  ==  should not be applied to functions
(e.g., for functions on numbers or lists, equality of functions is undecidable)

Solution: We introduce a type class \texttt{Eq} that contains all types whose elements can be checked for equality
(e.g., \texttt{Eq} contains \texttt{Int}, \texttt{Bool}, \texttt{[Int]}, ... but not \texttt{Int \to Int} etc.)

\((==), (\neq) :: \texttt{Eq}\ a \Rightarrow a \to a \to \texttt{Bool}\)

\[\text{Context}
\]

restricts the instantiation of the type \texttt{var} \(a\) to types from the type class \texttt{Eq}

\[\text{class Eq}\ a\ where\ \\texttt{\{c1, c2, \ldots, c\}}\]

\[\text{name of the new type class}
\]

\[\text{type variable, stands for the types of the new type class}
\]

\[\text{declarations which are used in all types of the new type class}
\]

\[\text{\texttt{class Eq}\ a\ where}\]

\[\texttt{(==), (\neq) :: a \to a \to \texttt{Bool}}\]

\[\text{\{default declaration that}\}
(==), (/=) :: a -> a -> Bool

x /= y = not (x == y)

\{ default declarations that can be overwritten by declarations in each instance type of the class Eq. \}

instance tyconstr instype where \{ idc1, ..., idc1 \}

means that the type instype is an instance/a member of the type tyconstr.

\{ declarations for this particular type instype which overwrite the default implementations of the class tyconstr \}

instance Eq Int where

(==) = prim Eq Int

pre-defined equality on integers

To evaluate

2 == 3 :: one uses prim Eq Int

2 /= 3 is not (2 == 3) (use default implementation of Eq)

Now we can define type classes and declare that certain types are members/instances of certain typeclasses.

⇒ We can now restrict any type by a context (Slide 20):

\begin{align*}
\text{Context:} & \quad (\text{tyconstr}_1 \, \text{var}_1, \ldots, \text{tyconstr}_n \, \text{var}_n) \\
\text{means that type variable } & \text{var}_i \text{ may only be instantiated}
\end{align*}
by a type from type class `tyconstr`.

Contexts cannot only occur in type declarations, but also in instance- and class-declarations.

**Contexts in instance declarations:**

The context means that `a` must be a type from the class `Eq`.

```haskell
instance Eq a => Eq [a] where

  []   == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _   == _       = False

  calls (=) on Eq a

  i.e., this is a recursive call

  for elements of type `a`
```

We have several implementations of `(==)`. It depends on the types of the arguments and the implementation is executed (ad-hoc polymorphism).

```haskell
instance (Eq a, Eq b) => Eq (a, b) where

  (x,y) == (x',y') = x == x' && y == y'
```

Haskell has several pre-defined type classes (like `Eq`). E.g., there is a type class `Show` that contains all types that can be converted to strings and shown on the screen.

```haskell
class Show a where
```
show :: a → String

To declare that a certain type is a member of the type class Show, up to now we added "deriving Show" to the corresponding data declaration.

This adds a corresponding instance declaration and default implementation for the functions of the class.

Similarly "deriving Eq" etc.

But we could also define "show" ourselves:

data List a = Nil | Cons a (List a)

instance Show a ⇒ Show (List a) where

  show Nil = " [] "
  show (Cons x xs) = show x ++ " :: " ++ show xs

Contexts in Class Declarations:

needed for a hierarchical organisation of typeclasses

Ex: Type class Ord for types whose elements can be ordered, i.e., here we have functions like

  > , < , ≥ , ≤

If a type is a member of Ord, then it should also be a member of Eq.
If a type is a member of \( \text{Eq} \) then it should also be a member of \( \text{Ord} \) :

\[
\Rightarrow \text{Ord should be a subclass of Eq}
\]

\[
\text{Context ensures that } a \text{ is from the class Eq, i.e., } \text{Ord } \subseteq \text{Eq}
\]

\[
\text{class Eq a } \Rightarrow \text{Ord a} \text{ where}
\]

\[
( >), ( <), ( >=), ( <=) :: a \rightarrow a \rightarrow \text{Bool}
\]

\[
x < y = x <= y \land \land x /= y
\]

Type classes are also useful for overloading arithmetic operators like \( +, -, *, \ldots \)

\[
\text{+ can be used for Int, Float, } \ldots
\]

\[
2 \quad :: \quad 2 + 3.5
\]

\[
\text{class (Eq a, Show a) } \Rightarrow \text{(Num a) where}
\]

\[
\vdots
\]

Type of 2 : \( \text{Num a } \Rightarrow a \)

i.e. 2 has any type a where

\( a \) is a member of the type class \( \text{Num} \)