第 04 章：类型与类簇

中英文对照：

- 类型 => type
- 类簇 => type class

✧ What is a type?

A type is a collection of related values.

For example, in Haskell the basic type Bool, contains two logical values: True, and False.

✧ Type Errors / 类型错误

“Applying a function to one or more arguments of the wrong type” is called a type error.

```
nrutas — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/lib/ghc-9.4.2/lib --interactive — 62×8
ghci> 1 + False

<interactive>:1:1: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the first argument of ‘(+)’, namely ‘1’
      In the expression: 1 + False
      In an equation for ‘it’: it = 1 + False
ghci>
```

- 1 is a number, and False is a logical value
- but + requires two numbers

✧ Types in Haskell

If evaluating an expression e would produce a value of type T, then e has type T, written

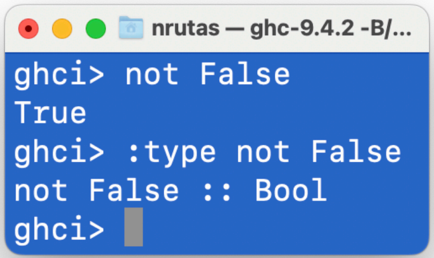
    e :: T

Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

```
   f :: A -> B,   e :: A
----------------------
        f e :: B
```

All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

In GHCi, the :type command calculates the type of an expression, without evaluating it.



- Basic Types in Haskell
  - Bool
    - logical values: True | False
    - exported by Prelude
  - Char
    - an enumeration whose values represent Unicode code points (i.e. characters, see http://www.unicode.org/ for details)
    - exported by Prelude
  - String
    - definition: type String = [Char]
    - exported by Prelude
  - Int
    - fix-precision integer numbers.
    - in GHC, the range of Int is [-2^63, 2^63-1]
    - exported by Prelude
  - Integer

        - arbitrary-precision integer numbers
        - exported by Prelude
    - Word
        - fix-precision unsigned integer numbers
        - the same size with Int
        - exported by Prelude
    - Natural
        - arbitrary-precision unsigned integer numbers
        - exported by Numeric.Natural (a module in the base package)
    - Float
        - single-precision floating-point numbers
        - exported by Prelude
    - Double
        - double-precision floating-point numbers
        - exported by Prelude

```
nrutas — ghc-9.4.2 -B/U...
ghci> sqrt 2 :: Float
1.4142135
ghci> sqrt 2 :: Double
1.4142135623730951
ghci>
```

- **List** Types

  A list is a sequence of values of the same type.

```
nrutas — ghc-9.4.2 -B/Users/nrutas/.ghcu...
ghci> :type [False, True, False]
[False, True, False] :: [Bool]
ghci> :type ['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd'] :: [Char]
ghci>
```

  Given a type T:

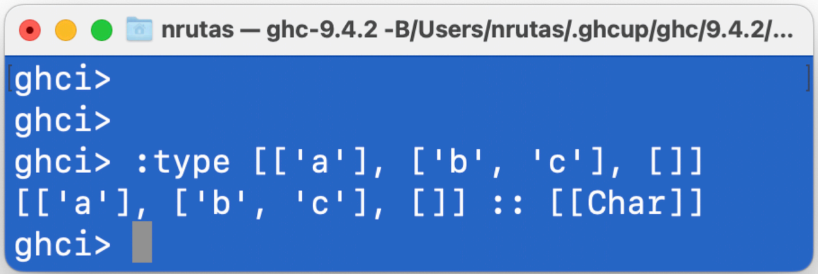
[T] is the type of of lists with elements of type T

Notes:

- The type of a list says nothing about the list's length.
- The type of the elements is unrestricted.

  That is, for any type T, [T] is a type of lists.

  For example, we can have lists of lists



✧ **Function** Types

A function is a mapping from values of one type to values of another

```
-- | Boolean \"not\"
not                          :: Bool -> Bool
not True                     =  False
not False                    =  True
```

Given two types X and Y:

X -> Y is the type of functions that map values of X to values of Y

Notes:

- The argument and result types are unrestricted

  For example, functions with multiple arguments or results are possible using lists or tuples.

```
add :: (Int, Int) -> Int
add (x,y) = x + y
```

```
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

✧ Curried functions

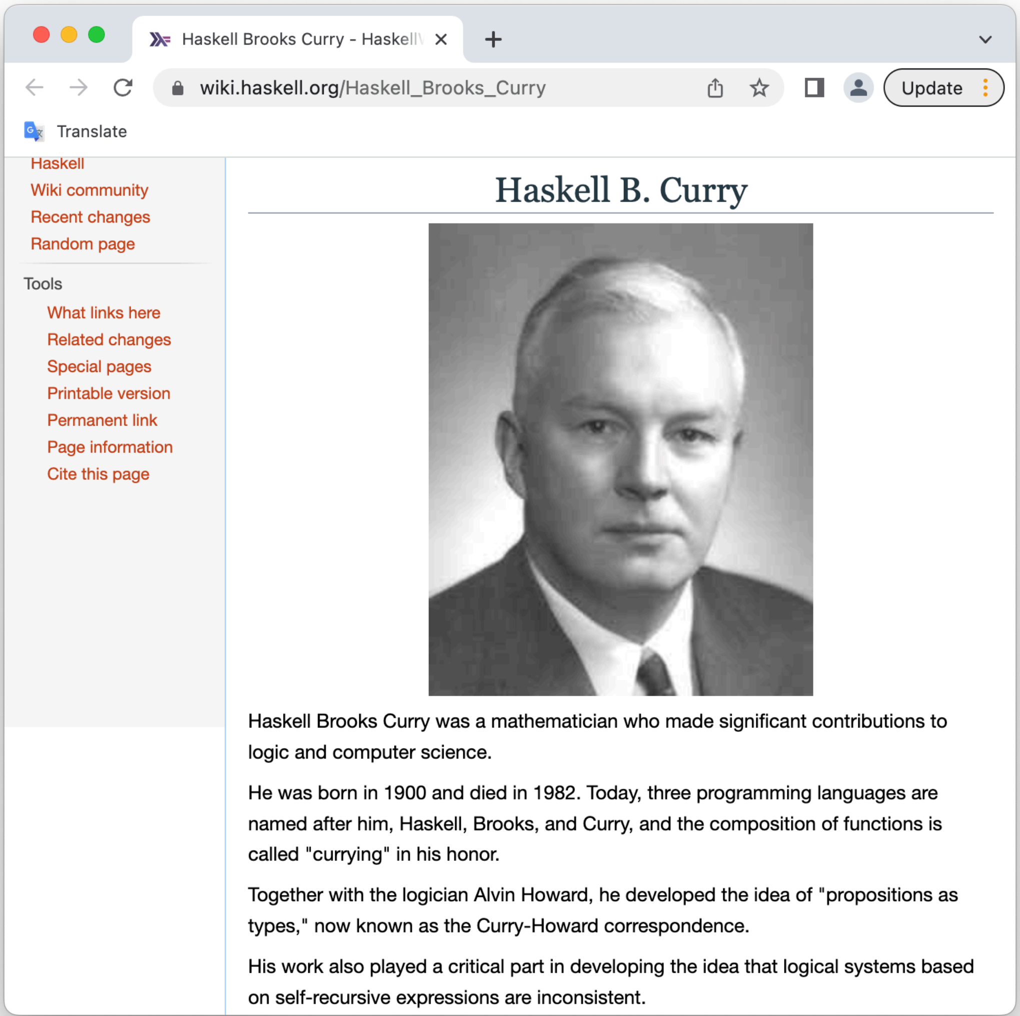
Functions with multiple arguments are also possible by returning functions as results.

```
add :: (Int, Int) -> Int
add (x, y) = x + y

add' :: Int -> Int -> Int
add' x y = x + y
```

- add' takes an integer x and returns a function add' x
  add' x takes an integer y and returns the result x + y
- add and add' produce the same final result
  but add takes its two arguments at the same time,
  whereas add' takes them one at a time

Functions that take their arguments one at a time
are called **curried functions**,
celebrating the work of Haskell Curry on such functions.



Functions with more than two arguments can be curried by returning nested functions.

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

- `mult x :: Int -> Int -> Int`
- `mult x y :: Int -> Int`
- `mult x y z :: Int`

✧ Why is Currying Useful?

Curried functions are more flexible than functions on tuples.

Useful functions can often be made by partially applying a curried function.

For example:

- add' 1 :: Int -> Int
- take 5 :: [Int] -> [Int]
- drop 5 :: [Int] -> [Int]

✧ Currying Conventions

The arrow -> associates to the right.

```
Int -> Int -> Int -> Int   ===   Int -> (Int -> (Int -> Int))
```

As a consequence, it is then natural for function application to associate to the left.

```
mult x y z   ===   ((mult x) y) z
```

Unless tupling is explicitly required,

all functions in Haskell are normally defined in curried form.

✧ Polymorphic Functions

A function is called polymorphic (“of many forms”)

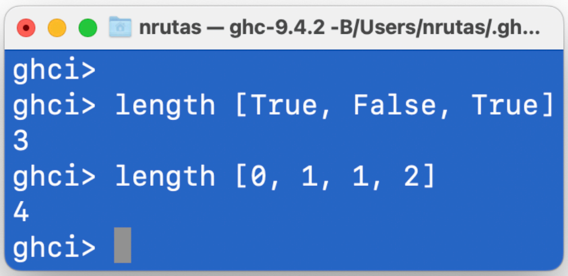
if its type contains one or more type variables.

```
length :: [a] -> Int
```

- For any type a,

  length takes a value of type [a], and returns a value of type Int

Type variables can be instantiated to different types in different circumstances:



- in **length [True, False, True]**, **a** is instantiated to **Bool**
- in **length [0, 1, 1, 2]**, **a** is instantiated to **Int**

Type variables must begin with a **lower-case** letter, and are usually named **a**, **b**, **c**, etc.

✧ Polymorphic Functions in Prelude : **examples**

```
fst :: (a, b) -> a
```

- Extract the fist component of a pair

```
snd :: (a, b) -> b
```

- Extract the second component of a pair

```
curry :: ((a, b) -> c) -> a -> b -> c
```

- Convert an uncurried function to a curried function
- Example:

    curry fst 1 2   ===   1

```
head :: [a] -> a
```

- Extract the first element of a list, which must be non-emmpty
- Example:

    head [1, 2, 3]   ===   1

    head [1..]   ===   1

    head []   throws an exception: Prelude.head: empty list

```
last :: [a] -> a
```

- Extract the last element of a list, which must be finite and non-empty
- Example:

    last [1, 2, 3]   ===   3

    last [1..]   hangs forever

    last []   throws an exception: Prelude.last: empty list

✧ Overloaded Functions

A polymorphic function is called overloaded, if its type contains one or more **type class constraints**.

```
program — ghc-9.4.2 -B/Users/nr...
ghci>
ghci> :type (+)
(+) :: Num a => a -> a -> a
ghci>
ghci>
```

- For any type **a** that is an instance of type class **Num**,
  (+) takes two values of type **a** and returns a value of type **a**.

Constrained type variables can be instantiated to any types that satisfy the constraints:

```
program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/lib/ghc-9.4.2/lib --interactive —...
ghci>
ghci> 1 + 2
3
ghci> 1.0 + 2.0
3.0
ghci> 'a' + 'c'

<interactive>:14:5: error:
    • No instance for (Num Char) arising from a use of '+'
    • In the expression: 'a' + 'c'
      In an equation for 'it': it = 'a' + 'c'
ghci>
```

- The error above is caused by the fact that

  the type **Char** is not an instance of type class **Num**.

✧ Type Class

Prelude exports many type classes, for example:

- **Eq / Ord / Num**

These type classes appear in many types of functions:

program — ghc-9.4.2 -B/Users/nrutas/.gh...

```
ghci>
ghci> :type (==)
(==) :: Eq a => a -> a -> Bool
ghci>
ghci> :type (<)
(<) :: Ord a => a -> a -> Bool
ghci>
ghci> :type (+)
(+) :: Num a => a -> a -> a
ghci>
```

✧ Type Class: **Eq**

```
class Eq a where
     (==), (/=) :: a -> a -> Bool
     x /= y      = not (x == y)
     x == y      = not (x /= y)
```

- The above is the definiton of type class **Eq**
- The **Eq** class defines equality (==) and inequality (/=)
- All basic datatypes exported by Prelude are instances of **Eq**
- **Eq** may be derived for any datatype whose constituents are also instances of **Eq**.

- The Haskell Report defines no laws for **Eq**
- However, instances are encouraged to satisfy the following properties:
  - Reflexivity / 自反性

    x == x   ===   True
  - Symmetry / 对称性

    x == y   ===   y == x
  - Transitivity / 传递性

    IF   x == y && y == z === True   THEN   x == z === True
  - Extensionality / 外延性

    IF x == y === True && f is a function whose return type is an instance of Eq THEN f x == f y === True

- Negation

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

- Minimal complete definition: (==) | (/=)

  If you want to make a type T an instance of Eq,

  you can only provide an implementation of one of the two funtcions (==) and (/=) on the type T.

✧ Type Class: **Ord**

```
data Ordering  =  LT | EQ | GT


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

    compare x y  =  if x == y then EQ
                else if x <= y then LT
                else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

- Ord, as defined by the Haskell report, implements a total order, and has the following properties:
  - Comparability

    x <= y || y <= x   ===   True
  - Transitivity

    IF   x <= y && y <= z === True   THEN   x <= z === True
  - Reflexivity

    x <= x   ===   True

- Antisymmetry

    IF x <= y && y <= x === True THEN x == y === True

- The following operator interactions are expected to hold:
  1. x >= y === y <= x
  2. x < y   === x <= y && x /= y
  3. x > y   === y < x
  4. x < y   === compare x y == LT
  5. x > y   === compare x y == GT
  6. x == y === compare x y == EQ
  7. min x y == if x <= y then x else y   ===   True
  8. max x y == if x >= y then x else y   ===   True

- Minimal complete definition: compare | (<=)

✧ Type Class: **Num**

```
class  Num a  where
    (+), (-), (*)       :: a -> a -> a

    -- Unary negation.
    negate              :: a -> a

    -- Absolute value.
    abs                 :: a -> a

    -- Sign of a number.
    signum              :: a -> a

    -- Conversion from an Integer.
    fromInteger         :: Integer -> a

    x - y     = x + negate y
    negate x = 0 - x
```

- The Haskell Report defines no laws for Num.
- However, (+) and (*) are customarily expected to define a ring

and have the following properties:

- Associativity of (+)

  (x + y) + z   ===   x + (y + z)

- Commutativity of (+)

  x + y   ===   y + x

- fromInteger 0 is the additive identity

  x + fromInteger 0   ===   x

- negate gives the additive inverse

  x + negate x   ===   fromInteger 0

- Associativity of (*)

  (x * y) * z   ===   x * (y * z)

- fromInteger 1 is the multiplicative identity

  x * fromInteger 1   ===   x

  fromInteger 1 * x   ===   x

- Distributivity of (*) with respect to (+)

  a * (b + c) === (a * b) + (a * c)

  (b + c) * a === (b * a) + (c * a)

○ Minimal complete definition:

(+), (*), abs, signum, fromInteger, (negate | (-))

作业 01

What are the types of the following values?

○ ['a', 'b', 'c']

○ ('a', 'b', 'c')

○ [(False, '0'), (True, '1')]

○ ([False, True], ['0', '1'])

○ [tail, init, reverse]

作业 02

What are the types of the following functions?

- `second xs = head (tail xs)`
- `swap (x, y) = (y, x)`
- `pair x y = (x, y)`
- `double x = x * 2`
- `palindrome xs = reverse xs == xs`
- `twice f x = f (f x)`

作业 03

阅读教科书，用例子（在 ghci 上运行）展示 Int 与 Integer 的区别以及 show 和 read 的用法。

作业 04

阅读教科书和 Prelude 模块文档，理解 Integral 和 Fractional 两个 Type Class 中定义的函数和运算符，用例子（在 ghci 上运行）展示每一函数/运算符的用法。