Functional Programming

Lecture 2 — Values, types and classes

Pedro Vasconcelos
DCC/FCUP

2025

Expressions and values

- Computation in functional programming is based on the evaluation of expressions
- You can think of evaluation as simplification using definitions and arithmetic operations
- Expressions that can't be further simplified are values

$$\overbrace{\text{sum}\ [1,2,3] = 1 + 2 + 3 = \underbrace{6}_{\text{value}}}^{\text{expressions}}$$

Types

A type is a name for a collection of related values.

For example, the Bool type contains the two logical values:

```
True        False
```

Type errors

Some operations only make sense with values of specific types.

For example: it is not possible to add together numbers and logical values.

```
> 1 + False
<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use
    • In the expression: 1 + False
      In an equation for ‘it’: it = 1 + False
```

In Haskell, such errors are detected by classifying expressions with the result type.

Type assignment

We write

```
e :: T
```

to state that the expression *e* admits a type *T*.

- If $e :: T$, then the result of evaluating *e* will be value of type *T*
- The interpreter or compiler checks types stated by the programmer and infers any omitted types
- Any type errors will be reported *before* attempting to run the program
- Programs with type errors can not be run

Basic types

Bool logical values

`True, False`

Char single Unicode characters

`'A','B','?','\n'`

String sequences of characters

`"Abacate", "UB40"`

Int fixed precision integers (typically 64-bits)

`142, -1233456`

Integer arbitrary precision integers
(only limited by the available memory)

Float single precision floating point numbers

`3.14154, -1.23e10`

Double double precision floating point numbers

Lists

A *list* is a sequence of elements all of the same type.

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

More generally: `[`*T*`]` is the type of lists whose elements are of type *T*.

Lists

A *list* is a sequence of elements all of the same type.

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

More generally: `[`*T*`]` is the type of lists whose elements are of type *T*.

Special case: `String` is equivalent to `[Char]`.

```
"abcd" == ['a','b','c','d']
```

Tuples

A *tuple* is either a pair, triple, quadruple etc. of elements with possibly distinct types.

```
(42,'a') :: (Int,Char)
(False,'b',True) :: (Bool,Char,Bool)
```

More generally: $(T_1, T_2, \ldots, T_n)$ is a tuple of n components of types T_i for i from 1 to n.

The Haskell standard ensures tuples up-to $n = 7$. In practice, tuples are usually of smaller sizes.

Observations

- Lists of different sizes *may* have identical types
- Tuples of different sizes *must* have different types

```
['a'] :: [Char]

['b','a','b'] :: [Char]

('a','b') :: (Char,Char)

('b','a','b') :: (Char,Char,Char)
```

Observations (cont.)

The elements of lists and tuples can be arbitrary, even lists and tuples themselves.

```
[['a'], ['b','c']] :: [[Char]]

(1,('a',2)) :: (Int,(Char,Int))

(1, ['a','b']) :: (Int,[Char])
```

Observations (cont.)

- The *empty list* `[]` admit any list type `[`T`]`
- The *empty tuple* `()` is the only expression of the unit type `()`
- There are no tuples of a single component: (v) is the same as v

Function types

A function maps values from one type to another.

```
not :: Bool → Bool

isDigit :: Char → Bool
```

- $A \to B$ is the type of functions mapping values of type A to values of type B
- The arrow symbol $\to$ is typed in the keyboard as `->`

Function types (cont.)

The argument and result of a function can be of any type.

```
add :: (Int,Int) → Int
add (x,y) = x+y

count :: Int → [Int]
count n = [0..n]
```

Functions with multiple arguments

A function with multiple arguments takes one argument at a time.

```
add :: Int → (Int → Int)
add x y = x+y

increment :: Int → Int
increment = add 1
```

Hence: `add 1` is a function that adds one to any integer.

This way of dealing with multiple arguments is named "*currying*" after Haskell Curry.[1]

[1] Although the original idea came from an earlier mathematician (Moses Schönfinkel).

Why use currying?

Curried functions are more flexible than ones with arguments are tuples because we can apply them partially.

```
take :: Int → [Char] → [Char]   -- get a prefix

take 5 :: [Char] → [Char] -- get prefix of 5 elms
```

Syntax conventions

The following two conventions reduce the amount of necessary parenthesis:

- arrows $\rightarrow$ associate to the *right*
- application associates to the *left*

```
    Int → Int → Int → Int
=   Int → (Int → (Int → Int))
```

```
   f x y z
= (((f x) y) z)
```

Polymorphic functions

Some functions can operate with values of any type; such functions admit types with *variables*.

A function with a type with variables is said to be polymorphic (meaning “of many forms”).

Example

```
length :: [a] → Int
```

The *length* function computes the length of a list of values of any type *a*.

Polymorphic functions (cont.)

When we apply a polymorphic function, the type variables are implicity replaced by the concrete types.

```
ghci> length [1,2,3,4]  -- Int
4
ghci> length "abacate"  -- Char
7
ghci> length [False,True] -- Bool
2
ghci> length [(2,'A'),(3,'C')] -- (Int,Char)
2
```

Type variables must start with a lower case letter; by conventions these are typically *a*, *b*, *c*...

Polymorphic functions (cont.)

Many Prelude functions are polymorphic.

```
null :: [a] → Bool

head :: [a] → a

take :: Int → [a] → [a]

fst :: (a,b) → a

zip :: [a] → [b] → [(a,b)]
```

Because of polymorphism we can use these functions in many contexts, including with new types defined by ourselves.

Overloading

A different kind of polymorphism occurs when a function operates on *some* types but not *all* types.

For example: `sum` works only with numeric types.

```
ghci> sum [1,2,3]
6
ghci> sum [1.5, 0.5, 2.5]
4.5
ghci> sum ['a', 'b', 'c']
  No instance for (Num Char) ...
ghci> sum [True, False]
  No instance for (Num Bool) ...
```

This is called overloading and is handled by type classes.

Overloading (cont.)

The most general type for `sum` is:

```
sum :: Num a ⇒ [a] → a
```

- The fat arrow ⇒ is typed in the keyboard as `=>`
- `Num a ⇒...` is a constraint on the type variable *a*
- It indicates that `sum` operates only on types *a* in the class `Num`
- All numeric types we have seen belong to this class: `Int`, `Integer`, `Float`, `Double`
- We also say that "`Int` implements the class `Num`" or "`Int` has a `Num` instance"

Some classes in the Prelude

Num	numeric types
Integral	numbers with integral division (div, mod)
Fractional	numbers with fractional division (/)
Eq	types with equality (==, /=)
Ord	types with total order (<, >, <=, >=)
Show	types which can be converted to strings
Read	types which can be converted from strings

Some classes in the Prelude (cont.)

We can inspect type classes and its instances in GHCi:

```
ghci>:info Num
class Num a where
  (+) :: a → a → a
  (-) :: a → a → a
  (*) :: a → a → a
  negate :: a → a
  abs :: a → a
  signum :: a → a
  fromInteger :: Integer → a
instance Num Double
instance Num Float
instance Num Int
instance Num Integer
instance Num Word
```

Type class hierarchy

Type classes can have hierarchical relations:

- `Ord` is a subclass of `Eq`
- `Fractional` and `Integral` are both subclasses of `Num`
- This means that:
  - every type that has an instance of `Ord` must also have an instance of `Eq`
  - every type that has an instance of `Fractional` must also have an instance of `Num`
  - every type that has an instance of `Integral` must also have an instance of `Num`

Mixing type classes

If we use overloaded functions from several classes, all of these will appear in the class constraints.

```
nub :: Eq a ⇒ [a] → [a]   -- remove duplicates
sum :: Num a ⇒ [a] → a    -- sum all values

combine :: (Eq a, Num a) ⇒ [a] → a
combine xs = sum (nub xs)
```

The order of type class constraints is not important, i.e. we could have written

```
combine :: (Num a, Eq a) ⇒ [a] → a
```

Numeric constants

In Haskell numeric constants can also be used at different types.

```
1 :: Int
1 :: Float
1 :: Num a ⇒ a -- most general type

3.0 :: Float
3.0 :: Double
3.0 :: Fractional a ⇒ a -- most general type
```

Hence, the following expression are well-typed.

```
1/3 :: Float
(1 + 1.5 + 2) :: Float
```

Mixing numeric types

Let us try to define a function to compute the average of a list of numbers.

```
average xs = sum xs / length xs
```

Mixing numeric types

Let us try to define a function to compute the average of a list of numbers.

```
average xs = sum xs / length xs
```

This looks correct, but it has a type error:

```
Could not deduce (Fractional Int) ...
```

Mixing numeric types (contd.)

The problem is that `/` can only be used with fractional numbers.

```
(/) :: Fractional a ⇒ a → a → a
                        -- fractional division
length xs :: Int        -- not fractional
```

The solution is to use an explicit convertion function.

```
average xs = sum xs / fromIntegral (length xs)
```

`fromIntegral` converts any integral type to some other numeric type:

```
fromIntegral :: (Integral a, Num b) ⇒ a → b
```

Note that both the argument and result are overloaded!

Type annotations

- Quite often we can write function definitions and let the type checker infer the most general types
- But the recommended practice is to always annotate types of top level functions:

```
average :: [Float] → Float
average xs = sum xs / fromIntegral(length xs)
```

- Benefits:
  - serves as code documentation
  - help to write the definitions
  - can sometimes help the compiler produce better error messages

Type annotations (cont.)

- Hint: start with a more concrete type before trying to generalize it
- Haskell will not let you write an incorrect type annotation!

```
average :: Num a ⇒ [a] → a          -- ERROR
average xs = sum xs / fromIntegral(length xs)

average :: Fractional a ⇒ [a] → a   -- OK
average xs = sum xs / fromIntegral(length xs)
```