Introduction to Functional Programming

Types, tuples, lists, and list comprehensions

Some slides are based on Graham Hutton's public slides

Recap previous lecture

GÖTEBORGS UNIVERSITET
CHALMERS

- Pattern matching
- Testing
- Gentle introduction to recursion

Today

GÖTEBORGS UNIVERSITET

CHALMERS

- Student representatives, please stay

- Continue with recursion
- Guarded equations (cases)

- Types
- Lists and tuples
- List comprehensions
- Properties

LIVE CODING!

TYPES

What is a type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False    True
```

GÖTEBORGS UNIVERSITET
CHALMERS

Type errors

- Applying a function to one or more arguments of the wrong type is called a type error.
- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.
  - Haskell is a strongly typed programming language
- Type checking catches many bugs (errors in your code) that would otherwise appear while running the code

```
ghci> 1 + False
error …
```

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

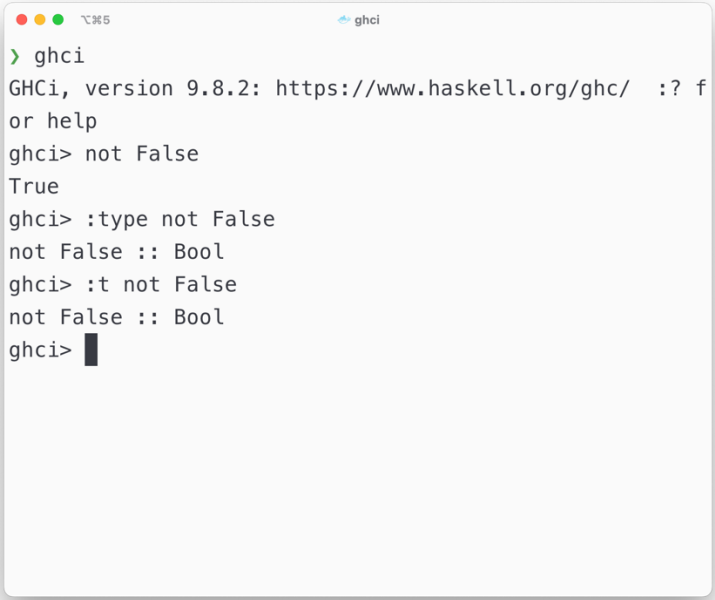
GÖTEBORGS UNIVERSITET
CHALMERS

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*.
- In GHCi, the `:type` command calculates the type of an expression, without evaluating it.
- We can annotate functions and expressions with their type using `::`



GÖTEBORGS UNIVERSITET

CHALMERS

Type	Values
`Integer`	integer numbers
`Int`	limited size integers
`Float, Double`	floating-point numbers
`Bool`	logical values
`Char`	single characters
`String`	strings of characters

Basic types

- All values in Haskell belong to a *type*
- To the left are some *basic* types
- You can define your own types

GÖTEBORGS UNIVERSITET
CHALMERS

List types

- A list is sequence of values of the *same type*
- In general:

  `[t]` is the type of lists with elements of type `t`

- The type of a list says nothing about its length

- The type of the elements is unrestricted. For example, we can have lists of lists.

```
[False,True,False] :: [Bool]

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


[False,True] :: [Bool]

[False,True,False] :: [Bool]


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

GÖTEBORGS UNIVERSITET
CHALMERS

Tuple types

- A tuple is a sequence of values of *different* types
- In general:

  $(t_1, t_2, \ldots, t_n)$ is the type of n-tuples whose ith components have type t_i for any i in 1...n

- The type of a tuple encodes its size

- The type of the components is unrestricted

```
(False,True) :: (Bool,Bool)
(False,'a',True) :: (Bool,Char,Bool)


(False,True) :: (Bool,Bool)
(False,True,False) :: (Bool,Bool,Bool)


('a',(False,'b')) :: (Char,(Bool,Char))
(True,['a','b']) :: (Bool,[Char])
```

GÖTEBORGS UNIVERSITET
CHALMERS

Function types

- A *function* is a mapping from values of one type to values of another type
- A function type describes what type of arguments (inputs) the function expects and what the type of the result (output) is
- In general:

  t_1 -> t_2 is the type of functions that map values of type t_1 to values to type t_2
- The argument and result types are unrestricted.
  - For example, functions with multiple arguments or, results are possible using lists or tuples

```
not :: Bool -> Bool
even :: Int -> Bool



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

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

GÖTEBORGS UNIVERSITET
CHALMERS

Overloaded functions

- An *overloaded* function is a function that can operate on values of *different* types
- We will explain this concept in detail later in the course
- For now, whenever you see a type like

  ```
  (+) :: Num a => a -> a -> a
  ```

  just think of it as a function that takes two arguments of type `a`, which must be a `Num`, and returns a value of type `a`

  In other words, the `(+)` operator works on *numeric* (`Num`) types

```
ghci> 1+2           -- a = Int
3

ghci> 1.0 + 2.0     -- a = Double
3.0

ghci> 'a' + 'b'     -- Char is not
error …             -- a numeric type
```

GÖTEBORGS UNIVERSITET

CHALMERS

Hint and tips

- When defining a new function in Haskell, it is useful to begin by writing down its type
- Within a script, it is good practice to state the type of every new function defined

LIST COMPREHENSIONS

GÖTEBORGS UNIVERSITET
CHALMERS

List comprehensions

- In mathematics, the *comprehension* notation can be used to construct new *sets* from old sets:

$$\{x^2 | \, x \in \{1 \ldots 5\}\}$$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1...5}

- In Haskell, a similar comprehension notation can be used to construct new *lists* from old lists:

```
[x^2 | x <- [1..5]]
```

- We can use list comprehensions to do something with every element in a list, and return the result in a new list

The list [1,4,9,16,25] of all numbers `x^2` such that `x` is an element of the list [1..5]

GÖTEBORGS UNIVERSITET
CHALMERS

List comprehensions

- The expression `x <- [1..5]` is called a *generator*, as it states how to generate values for `x`.
- List comprehensions can have *multiple* generators, separated by commas.
- Changing the *order* of the generators changes the order of the elements in the final list.
- Multiple generators are like *nested loops*, with later generators as more deeply nested loops whose variables change value more frequently.

```
ghci> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

ghci> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

`x <- [1,2,3]` is the last generator, so the value of the `x` component of each pair changes most frequently

GÖTEBORGS UNIVERSITET
CHALMERS

Dependant generators

- Later generators can *depend* on the variables that are introduced by earlier generators.

The list
`[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]`
of all pairs of numbers `(x,y)` **such that** `x,y` **are elements of the list** `[1..3]` **and** `y >= x`

- Using a dependant generator we can define the library function that *concatenates* a list of lists:

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

```
concat xss = [x | xs <- xss, x <- xs]

ghci> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

GÖTEBORGS UNIVERSITET
CHALMERS

Guards

- List comprehensions can use *guards* to restrict the values produced by earlier generators.
- We can use these guards to filter elements from a list.
- Just as generators, we can have multiple guards, which act as a conjunction.

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

```
[x | x <- [1..10], even x]

[x | x <- [1..100], even x, x < 50]
```

All elements between 1 and 100, which are even and smaller than 50

GÖTEBORGS
UNIVERSITET

CHALMERS