GÖTEBORGS UNIVERSITET

CHALMERS

Introduction to Functional Programming

Polymorphic functions and overloaded functions

Some slides are based on Graham Hutton’s public slides

Recap previous lecture

- Modelling with data types
- The ‘cons’ operator
- Defining (recursive) functions over lists

- Announcements:
  - All lab 1 submissions are graded 😅

Today

- Polymorphic functions
  - Type variables
- The `Maybe` data type
- Common type classes:
  - `Show`, `Eq`, `Ord`, `Num`

- Import declarations
- `where`-clauses and `let`-expressions

- (If time allows: QuickCheck)

LIVE CODING!

Polymorphic functions

- A function is called *polymorphic* ("of many forms") if its type contains one or more *type variables*.
- Type variables can be instantiated to different types in different circumstances.
- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

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

For any type `a`, length takes a list of values of type `a` and returns an integer

```
ghci> length [False, True]
2

ghci> length [1,2,3,4]
4
```

`a = Bool`

`a = Int`

Polymorphic functions

- Many of the functions defined in the standard prelude are polymorphic.

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

head :: [a] -> a

take :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a, b)]

id :: a -> a
```

Overloaded functions

- A polymorphic function is called *overloaded* if its type contains one or more *class constraints*.
- Constrained type variables can be instantiated to any types that satisfy the constraints.

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

For any numeric type a, (+) takes two values of type a and returns a value of type a

```
ghci> 1 + 2
3

ghci> 1.0 + 2.0
3.0

ghci> 'a' + 'b'
ERROR
```

a = Int

a = Double

Char is not a numeric type

Overloaded functions

- Haskell has a number of type classes, including:
  - `Num` – numeric types
  - `Eq` – equality types
  - `Ord` – ordered types
  - `Show` – showable types

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

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

(<)  :: Ord a => a -> a -> Bool

show :: Show a => a -> String
```

Hints and tips

- When defining a new function in Haskell, it is useful to begin by writing down its type.
- In a source code file, it is good practice to state the type of every new function defined.
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Strings are lists!

- A *string* is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.
- Because strings are just special kinds of lists, any *polymorphic* function that operates on lists can also be applied to strings.
- Similarly, list comprehensions can also be used to define functions on strings,
  - See the example on the right, which counts how many times a character occurs in a string

Means `[‘a′,′b′,′c′] :: [Char]`

```
″abc″ :: String
```

```
ghci> length "abcde"
5

ghci> take 3 "abcde"
"abc″
```

```
count :: Char -> String -> Int
count c s = length [x | x <- s, x == c]

ghci> count ′s′ ″Mississippi″
4
```

UNIVERSITAS GOTHOBURGENSIS 1891

GÖTEBORGS
UNIVERSITET

AVANCEZ 1829

CHALMERS