Functional Programming
Starting Haskell

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

Let's get started!

Haskell Demo

- Let's say we want to buy a game in the USA and we have to convert its price from USD to EUR
- A **definition** gives a name to a value
- Names are case-sensitive, must start with lowercase letter
- Definitions are put in a text file ending in `.hs`

Examples.hs

```
dollarRate = 1.3671
```

Using the definition

- Start the Haskell interpreter GHCi

  ```
  > stack ghci
  Configuring GHCi with the following packages:
  GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for h
  Loaded GHCi configuration from /private/var/folders/t4/skr
  Prelude>
  ```

- Load the file

  ```
  Prelude> :l Examples.hs
  [1 of 1] Compiling Main             ( Examples.hs, interpr
  Ok, modules loaded: Main.
  *Main>
  ```

- Use the definition

  ```
  *Main> dollarRate
  1.3671
  *Main> 53 * dollarRate
  72.4563
  ```

A function to convert EUR to USD

Examples.hs

```
dollarRate = 1.3671

-- |convert EUR to USD
usd euros = euros * dollarRate
```

- line starting with `--`: comment
- `usd`: function name (defined)
- `euros`: argument name (defined)
- `euros * dollarRate`: expression to compute the result

Using the function

- load into GHCi
  - as before or
  - use :r to reload

```
*Main> usd 1
1.3671
*Main> usd 73
99.7983
```

Converting back

Write a function `euro` that converts back from USD to EUR!

```
*Main> euro (usd 73)
73.0
*Main> euro (usd 1)
1.0
*Main> usd (euro 100)
100.0
```

Converting back

Write a function euro that converts back from USD to EUR!

```
*Main> euro (usd 73)
73.0
*Main> euro (usd 1)
1.0
*Main> usd (euro 100)
100.0
```

Your turn

Testing properties
Is this function correct?

A reasonable property of euro and usd

```
prop_EuroUSD x = euro (usd x) == x

== is the equality operator

*Main> prop_EuroUSD 79
True
*Main> prop_EuroUSD 1
True
```

Testing properties
Is this function correct?

A reasonable property of euro and usd

```
prop_EuroUSD x = euro (usd x) == x

== is the equality operator

*Main> prop_EuroUSD 79
True
*Main> prop_EuroUSD 1
True
```

Does it hold generally?

Aside: Writing Properties

Convention

Function names beginning with prop_ are properties we expect to be True

Writing properties in a file

- Tells us how functions should behave
- Tells us what has been tested
- Lets us repeat tests after changing a definition

Testing

At the beginning of Examples.hs

```
import Test.QuickCheck
```

A widely used Haskell library for automatic random testing

Testing

At the beginning of Examples.hs

```
import Test.QuickCheck
```

A widely used Haskell library for automatic random testing

May need to install it first . . .

```
stack install QuickCheck
```

Running tests

```
*Main> quickCheck prop_EuroUSD
*** Failed! Falsifiable (after 10 tests and 1 shrink):
7.0
```

- Runs 100 randomly chosen tests
- Result: The property is wrong!
- It fails for input 7.0

Running tests

```
*Main> quickCheck prop_EuroUSD
*** Failed! Falsifiable (after 10 tests and 1 shrink):
7.0
```

- Runs 100 randomly chosen tests
- Result: The property is wrong!
- It fails for input 7.0

Check what happens for 7.0!

What happens for 7.0

```
*Main> usd 7
9.5697
*Main> euro 9.5697
6.999999999999999
```

The Problem: Floating Point Arithmetic

- There is a tiny difference between the initial and final values

  ```
  *Main> euro (usd 7) - 7
  -8.881784197001252e-16
  ```

- Calculations are only performed to about 15 significant figures
- The property is wrong!

Fixing the problem

- NEVER use equality with floating point numbers!
- The result should be *nearly* the same
- The difference should be small – smaller than 10E-15

Comparing Values

```
*Main> 2<3
True
*Main> 3<2
False
```

Defining "Nearly Equal"

- Can define new operators with names made up of symbols

In Examples.hs

```
x ~== y = x - y < 10e-15
```

```
*Main> 3 ~== 3.0000001
True
*Main> 3 ~== 4
True
```

Defining "Nearly Equal"

- Can define new operators with names made up of symbols

In Examples.hs

```
x ~== y = abs(x - y) < 10e-15 * abs x
```

```
*Main> 3 ~== 3.0000001
True
*Main> 3 ~== 4
True
```

Fixing the property

In Examples.hs
```
prop_EuroUSD' x = euro (usd x) ~== x
```

```
*Main> prop_EuroUSD' 3
True
*Main> prop_EuroUSD' 56
True
*Main> prop_EuroUSD' 7
True
```

Name the price

Let's define a name for the price of the game we want in Examples.hs

```
price = 79
```

Name the price

```
price = 79
```

After reload: Ouch!

```
*Main> euro price

<interactive>:57:6:
    Couldn't match expected type 'Double' with actual type 'In
    In the first argument of 'euro', namely 'price'
    In the expression: euro price
    In an equation for 'it': it = euro price
```

Every Value has a type

The :i command prints information about a nae

```
*Main> :i price
price :: Integer
   -- Defined at ...
*Main> :i dollarRate
dollarRate :: Double
   -- Defined at ...
```

More types

```
*Main> :i True
data Bool = ... | True  -- Defined in 'GHC.Types'
*Main> :i False
data Bool = False | ...  -- Defined in 'GHC.Types'
*Main> :i euro
euro :: Double -> Double
   -- Defined at...
*Main> :i prop_EuroUSD'
prop_EuroUSD' :: Double -> Bool
   -- Defined at...
```

- True and False are **data constructors**

Types matter

- Types determine how computations are performed
- A type annotation specifies which type to use

```
*Main> 123456789*123456789 :: Double
1.524157875019052e16
*Main> 123456789*123456789 :: Integer
15241578750190521
```

- `Double`: double precision floating point
- `Integer`: exact computation
- GHCi must know the type of each expression before computing it.

Type checking

- Infers (works out) the type of every expression
- Checks that all types match — before running the program

Our example

```
*Main> :i price
price :: Integer
   -- Defined at...
*Main> :i euro
euro :: Double -> Double
   -- Defined at...
*Main> euro price

<interactive>:70:6:
    Couldn't match expected type 'Double' with actual type 'In
    In the first argument of 'euro', namely 'price'
    In the expression: euro price
    In an equation for 'it': it = euro price
```

Why did it work before?

- Numeric literals are **overloaded**
- Giving the number a name fixes its type

```
*Main> euro 79
57.78655548240802
*Main> 79 :: Integer
79
*Main> 79 :: Double
79.0
*Main> price :: Integer
79
*Main> price :: Double

<interactive>:76:1:
    Couldn't match expected type 'Double' with actual type 'Ir
    In the expression: price :: Double
    In an equation for 'it': it = price :: Double
```

Fixing the problem

A definition can be given a **type signature** which specifies its type

In Examples.hs

```
-- |price of the game in USD
price' :: Double
price' = 79
```

```
*Main> :i price'
price' :: Double
    -- Defined at...
*Main> euro price'
57.78655548240802
```

Questions?