Functional Programming

Tutorial 1: Evaluating expressions

Twan van Laarhoven

7 September 2022

What is the Tutorial for?

- Bridge between lecture and practicals
- Asking your questions
- Getting ready for practicals by making exercises together

Hints

- Download templates from Brightspace
- Start ghci with your `.hs` files.

Questions?

Evaluating expressions

Given the following definitions:

```
square :: Integer → Integer
square x = x ∗ x

smallest :: Integer → Integer → Integer
smallest x y = if x ≤ y then x else y
```

Evaluate: smallest (square 2) (square 3).

Evaluating expressions

Given the following definition:

```
compute :: Integer → Integer → Integer
compute x y = let a = x − y
                  b = x + y
              in (a + 1) ∗ (b + 2)
```

Evaluate: compute 4 2.

Evaluating twice

```
double x = x * x
twice f x = f (f x)
```

a) Evaluate `twice double 3` by hand.

Evaluating twice

```
double x = x * x
twice f x = f (f x)
```

a) Evaluate `twice double 3` by hand.

b) What is the type of `twice`?

Evaluating twice

```
double x = x * x
twice f x = f (f x)
```

a) Evaluate `twice double 3` by hand.

b) What is the type of `twice`?

c) Why does this allow that twice can be applied to itself?

Evaluating with GHCi

Twice as a lambda expression

An alternative definition of `twice` builds on λ-expressions.

```
twice' = \f → \x → f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (* 2) 1`.

Twice as a lambda expression

An alternative definition of twice builds on λ-expressions.

```
twice' = \f → \x → f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (* 2) 1`.
For evaluating `twice'` we need to apply the evaluation rule for λ-expressions (the β-rule).

```
   (\x → body) arg  ⇒  body {x := arg}
```

Twice as a lambda expression

An alternative definition of `twice` builds on λ-expressions.

```
twice' = \f → \x → f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (* 2) 1`.
For evaluating `twice'` we need to apply the evaluation rule for λ-expressions (the β-rule).

```
   (\x → body) arg  ⇒  body {x := arg}
```

An example of applying this rule:

```
(\ x → x + x) 47
```

Twice as a lambda expression

An alternative definition of `twice` builds on λ-expressions.

```
twice' = \f → \x → f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (* 2) 1`.
For evaluating `twice'` we need to apply the evaluation rule for λ-expressions (the β-rule).

```
   (\x → body) arg   ⇒   body {x := arg}
```

An example of applying this rule:

```
(\ x → x + x) 47
=> x + x where x = 47
```

Twice as a lambda expression

An alternative definition of `twice` builds on λ-expressions.

```
twice' = \f -> \x -> f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (* 2) 1`.
For evaluating `twice'` we need to apply the evaluation rule for λ-expressions (the β-rule).

```
  (\x -> body) arg  =>  body {x := arg}
```

An example of applying this rule:

```
(\ x -> x + x) 47
=> x + x where x = 47
=> 47 + 47
```

Twice as a lambda expression

An alternative definition of `twice` builds on λ-expressions.

```
twice' = \f → \x → f (f x)
```

a) Evaluate `twice' (+ 1) 0` and `twice' twice' (∗ 2) 1`.
For evaluating `twice'` we need to apply the evaluation rule for λ-expressions (the β-rule).

```
  (\x → body) arg  ⇒  body {x := arg}
```

An example of applying this rule:

```
(\ x → x + x) 47
=> x + x where x = 47
=> 47 + 47
=> 94
```