technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1 Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)

Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung Übungsblatt 1

Ausgabe: 13.10.2017, **Abgabe:** 20.10.2017 – 16:00 Uhr, **Block:** 1

> Das `case`-Konstrukt wird im Abschnitt *Gleichungen, die Funktionen definieren* auf den Folien 19–22 vorgestellt. Bitte lesen Sie diesen Abschnitt selbstständig.

Hinweis: Um dieses Übungsblatt zu lösen, sind folgende Äquivalenzen hilfreich:

`x ⊗ y` ⇔ `(⊗) x y`	(Operator als Funktion)
`f x y` ⇔ ``x `f` y``	(Funktion als Operator)
`\x -> ... \z -> e` ⇔ `\x ... z -> e`	(λ-Ausdrücke zusammenfassen)
`f = \x ... z -> e` ⇔ `f x ... z = e`	(Applikative Definition)

(Patternmatching Umformung)

```
f x1 ... z1 | g1 = e1   ⇔ f x ... z =
           ⋮                case (x, ..., z) of
f xN ... zN | gN = eN         (x1,...,z1) | g1 -> e1
                                        ⋮
                              (xN,...,zN) | gN -> eN
```

Aufgabe 1.1 (3 Punkte) *Typeinferenz*
Berechnen Sie die Typen der folgenden Ausdrücke mithilfe der Typinferenzregeln.

a) `(\(x,y) -> Just 3) (3,3)` mit `3 :: Int`

b) `\f g x -> f (g x)`

Lösungsvorschlag

a) `(\(x,y) -> Just 3) (3,3)`

$$\dfrac{\dfrac{\dfrac{\texttt{x :: a,}\quad \texttt{y :: b}}{\texttt{(x,y) :: (a,b),}}\qquad \dfrac{\texttt{3 :: Int}}{\texttt{Just 3 :: Maybe Int}}}{\texttt{\textbackslash(x,y) -> Just 3 :: (a,b) -> Maybe Int,}}\qquad \dfrac{\texttt{3 :: Int,}\quad \texttt{3 :: Int}}{\texttt{(3,3) :: (Int,Int)}}}{\texttt{(\textbackslash(x,y) -> Just 3) (3,3) :: Maybe Int}}$$

b) `\f g x -> f (g x)` ⇔ `\f -> (\g -> (\x -> f(g(x))))`

$$\dfrac{\texttt{f :: b -> c,}\qquad \dfrac{\texttt{g :: a -> b,}\qquad \dfrac{\texttt{x :: a,}\qquad \dfrac{\texttt{f :: b -> c,}\qquad \dfrac{\texttt{g :: a -> b}\quad \texttt{x :: a}}{\texttt{g(x) :: b}}}{\texttt{f(g(x)) :: c}}}{\texttt{\textbackslash x -> f(g(x)) :: a -> c}}}{\texttt{\textbackslash g -> (\textbackslash x -> f(g(x))) :: (a -> b) -> a -> c}}}{\texttt{\textbackslash f -> (\textbackslash g -> (\textbackslash x -> f(g(x)))) :: (b -> c) -> (a -> b) -> a -> c}}$$

Aufgabe 1.2 (3 Punkte) *λ-Ausdrücke Auswerten*
Werten Sie folgende Ausdrücke schrittweise aus.

a) `(\x y -> x * y) 3 2`

b) `(\f g x -> f (g x)) (\y -> y * 2) (\z -> z + 1)`

Lösungsvorschlag

a)

```
(\x y -> x * y) 3 2
<=> (\x -> \y -> x * y) 3 2
~> (\y -> 3 * y) 2
~> 3 * 2 ~> 6
```

b)

```
(\f g x -> f (g x)) (\y -> y * 2) (\z -> z + 1)
<=>(\f -> \g -> \x -> f (g x)) (\y -> y * 2) (\z -> z + 1)
~>(\g -> \x -> (\y -> y * 2) (g x)) (\z -> z + 1)
~>\x -> (\y -> y * 2) ((\z -> z + 1) x)
~>\x -> (\y -> y * 2) (x + 1)
~>\x -> (x + 1) * 2
```

Aufgabe 1.3 (3 Punkte) *Haskell-Funktion Auswerten*
Gegeben sei folgende Haskell-Funktion:

```
and' :: Bool -> Bool -> Bool
and' False _ = False
and' True  b = b
```

Werten Sie den Ausdruck `and' True True` aus, indem Sie erst `and'` in einen λ-Ausdruck umformen und dann schrittweise auswerten.

Lösungsvorschlag
Als λ-Ausdruck:

```
and' :: Bool -> Bool -> Bool
and' False _ = False
and' True  b = b
<=>
and' b1 b2 = case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
<=>
and' = \b1 b2 -> case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
<=>
and' = \b1 -> \b2 -> case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
```

Auswertung:

```
and' True True
~> (\b1 -> \b2 -> case (b1,b2) of
      (False,_) -> False
      (True,b) -> b) True True
```

```
~> (\b2 -> case (True,b2) of
      (False,_) -> False
      (True,b) -> b) True
~> case (True,True) of
     (False,_) -> False
     (True,b) -> b
~> True
```

Aufgabe 1.4 (3 Punkte) *Haskell-Funktionen definieren*
Schreiben Sie eine Haskell-Funktionen `ite` vom Typ `Bool -> Int -> Int -> Int`, welche die erste Ganzzahl (`Int`) zurückgibt, falls der erste Parameter vom Wert `True` ist und die zweite Ganzzahl sonst.
Beispiele:

```
ite True 3 9 ~> 3
ite False 3 9 ~> 9
```

a) Definieren Sie die Funktion als λ-Ausdruck mit `case`.

b) Definieren Sie die Funktion applikativ.

Definieren Sie die Funktionen ohne `if_then_else_`-Ausdruck.

Lösungsvorschlag

a) λ-Ausdruck mit `case`:

```
ite = \b -> \x -> \y -> case (b,x,y) of
  (True,x,_) -> x
  (False,_,y) -> y
```

Einfacher:

```
ite = \b -> \x -> \y -> case b of
  True -> x
  False -> y
```

b) applikativ:

```
ite True  x _ = x
ite False _ y = y
```