technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2016/2017

# Übungen zu Funktionaler Programmierung
# Übungsblatt 7

**Ausgabe:** 25.11.2016, **Abgabe:** 2.12.2016

**Aufgabe 7.1** (3 Punkte)

1. Definieren Sie den Typ `Person` mit den Attributen (Destruktoren) `name`, `familyName`, `age`. Benutzen Sie sinnvolle Typen für die Attribute.
2. Schreiben Sie die Funktionen `getX` und `setX` aus der Aufgabe 6.3 so um, dass ausschließlich Destruktoren verwendet werden.

Der Datentyp `Point` sei jetzt wie folgt definiert:

```
data Point = Point{ x:: Double, y :: Double }
```

**Lösungsvorschlag**

```
data Person = Person{ name :: String, familyName :: String, age :: Int }

getX :: Point -> Double
getX = x

setX :: Double -> Point -> Point
setX x pt = pt{x = x}
```

**Aufgabe 7.2** (3 Punkte)

1. Stellen Sie den Ausdruck $3x^2 + 2y + 1$ als Element vom Typ `Exp String` da.
2. Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie gebrauch von `exp2store`.

**Lösungsvorschlag**

1.
```
expr :: Exp String
expr = Sum [3 :* (Var "x" :^ 2), 2 :* Var "y", Con 1]

solutions :: [(Int,Int,Int)]
solutions = [(x,y,z) | z <- [0..], x <- [0..z], y <- [0..z]
            , exp2store expr (st x y) == z
            ]
  where
    st x y "x" = x
    st x y "y" = y
```

**Aufgabe 7.3** (3 Punkte) Schreiben Sie eine Funktion `simplify :: Exp x -> Exp x`, welche arithmetische Ausdrücke vereinfacht. Dabei sollen folgende Gleichungen zur Vereinfachung genutzt werden:

$$0 + x = x \qquad x^0 = 1$$
$$0 * x = 0 \qquad x^1 = x$$
$$1 * x = x$$

**Lösungsvorschlag**

```
simplify :: Exp x -> Exp x
-- Potenzen
simplify (e :^ 0) = Con 1
simplify (e :^ 1) = simplify e
simplify (e :^ i) = simplify e :^ i
-- Summen
simplify (Sum []) = Con 0
simplify (Sum es) = Sum $ filter (conEqInt 0) $ map simplify es
-- Produkte
simplify (Prod []) = Con 1
simplify (Prod es)
  | any (conEqInt 0) simplified = Con 0
  | otherwise  = Prod $ filter (conEqInt 1) simplified
  where simplified = map simplify es
simplify (i :* e) = simplMul (i :* simplify e)
  where
    simplMul (0 :* _) = Con 0
    simplMul (_ :* Con 0) = Con 0
    simplMul (1 :* e) = e
    simplMul (i :* Con 1) = Con i
    simplMul e = e
-- Sonstiges
simplify (e1 :- e2) = simplify e1 :- simplify e2
simplify e = e

conEqInt :: Int -> Exp x -> Bool
conEqInt i (Con c) = i == c
conEqInt _ _       = False
```

**Aufgabe 7.4** (3 Punkte) Schreiben Sie eine Funktion `bexp2store`, welche sich ähnlich wie `exp2store` verhält. Anstelle arithmetischer Ausdrücke sollen boolesche Ausdrücke vom Typ `BExp x` ausgewertet werden. Diese Funktion benötigt zwei Variablenbelegungen. Eine für boolesche Ausdrücke und die andere für arithmetische Ausdrücke.

Benutzen Sie folgende Typen:

```
type BStore x = x -> Bool
bexp2store :: BExp x -> BStore x -> Store x -> Bool
```

**Lösungsvorschlag**

```
bexp2store :: BExp x -> BStore x -> Store x -> Bool
bexp2store True_ _ _ = True
bexp2store False_ _ _ = False
bexp2store (BVar x) bst _ = bst x
```

```
bexp2store (Or bs) bst st = or $ map (\x -> bexp2store x bst st) bs
bexp2store (And bs) bst st = and $ map (\x -> bexp2store x bst st) bs
bexp2store (Not bs) bst st = not $ bexp2store bs bst st
bexp2store (e1 := e2) _ st = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) _ st = exp2store e1 st <= exp2store e2 st
```