technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 6

Ausgabe: 24.11.2017, **Abgabe:** 1.12.2017 – 16:00 Uhr, **Block:** 3

Aufgabe 6.1 (3 Punkte) *Arithmetische Ausdrücke*

a) Stellen Sie den Ausdruck $2x^3 + 5y + 2$ und z^2 als Elemente vom Typ `Exp String` da.

b) Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie sinvollen gebrauch von den beiden Ausdrücken und `exp2store`.

```
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z)
  | z <- [0..]
  , y <- [0..z^2]
  , x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2
  ]
```

Hinweis: Importieren Sie das Modul `Expr`.

Lösungsvorschlag

a)
```
expr1, expr2 :: Exp String
expr1 = Sum [2 :* (Var "x" :^ 3), 5 :* Var "y", Con 2]
expr2 = Var "z" :^ 2
```

b)
```
solutions :: [(Int,Int,Int)]
solutions = [(x,y,z)
  | z <- [0..], x <- [0..z^2], y <- [0..z^2]
  , let st "x" = x
        st "y" = y
        st "z" = z
  , exp2store expr1 st == exp2store expr2 st
  ]
```

Aufgabe 6.2 (3 Punkte) *Boolesche Ausdrücke*
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 -> Store x -> BStore x -> Bool
```

Hinweis: Importieren Sie das Modul `Expr`.

Lösungsvorschlag

```
bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store True_ _ _ = True
bexp2store False_ _ _ = False
bexp2store (BVar x) _ bst = bst x
bexp2store (Or bs) st bst = or $ map (\x -> bexp2store x st bst) bs
bexp2store (And bs) st bst = and $ map (\x -> bexp2store x st bst) bs
bexp2store (Not bs) st bst = not $ bexp2store bs st bst
bexp2store (e1 := e2) st _ = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) st _ = exp2store e1 st <= exp2store e2 st
```

Aufgabe 6.3 (3 Punkte) *Typklassen*
Schreiben Sie eine Klasse für eine überladene Funktion `drop'`. Diese soll sich wie `drop` verhalten, aber nicht auf den Listentyp `[a]` beschränkt sein. Instanziieren Sie die Klasse für `[a]`, `Colist a` und `Stream a`.

Lösungsvorschlag

```
class Drop a where
  drop' :: Int -> a -> a

instance Drop [a] where
  drop' = drop

instance Drop (Colist a) where
  drop' 0 s = s
  drop' n ls = case split ls of
    Just (_,s) | n > 0 -> drop' (n-1) s
    Nothing            -> Colist Nothing

instance Drop (Stream a) where
  drop' 0 s = s
  drop' n (_:<s) | n > 0 = drop' (n-1) s
```

Aufgabe 6.4 (3 Punkte) *Binäre Bäume*
Gegeben seien folgende Datentypen:

```
data Bintree a = Empty | Fork a (Bintree a) (Bintree a) deriving Show
data Edge = Links | Rechts deriving Show
type Node = [Edge]
```

Definieren Sie folgende Funktionen.

a) `value :: Node -> Bintree a -> Maybe a` – Gibt den Wert des Knoten zurück. Falls der Knoten nicht existiert, wird `Nothing` ausgegeben.

b) `search :: Eq a => a -> Bintree a -> Maybe Node` – Durchsucht den Baum nach dem angegebenen Wert. Falls Knoten mit dem Wert existieren, wird der Erste ausgegeben. Ansonsten wird `Nothing` zurückgegeben. Die Suchreihenfolge ist beliebig.

Lösungsvorschlag

a)
```
value :: Node -> Bintree a -> Maybe a
value (Rechts:nodes) (Fork _ _ r) = value nodes r
value (Links:nodes) (Fork _ l _) = value nodes l
value [] (Fork a _ _) = Just a
value _ Empty = Nothing
```

b)
```
search :: Eq a => a -> Bintree a -> Maybe Node
search _ Empty = Nothing
search a (Fork b l r)
  | a == b    = Just []
  | otherwise = f (search a l) (search a r) where
      f (Just l) _ = Just $ Links:l
      f Nothing (Just r) = Just $ Rechts:r
      f _ _ = Nothing
```