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 12

Ausgabe: 20.1.2016, **Abgabe:** 27.1.2017 - 12:00 Uhr

Aufgabe 12.1 (2 Punkte) Gegeben sei folgende Listenkomprehension:

```
solutions :: [(Int , Int , Int )]
solutions = [ (x,y,z) | z <- [0..] , y <- [0..z] , x <- [0..z]
            , 3*x^2 + 2*y + 1 == z]
```

1. Überführen Sie die Listenkomprehension in die do-Notation.
2. Überführen Sie die do-Notation in monadische Operatoren und Funktionen(»=,»,`return`).

Lösungsvorschlag

1. do-Notation

```
solutions' :: [(Int , Int , Int )]
solutions' = do
   z <- [0..]
   y <- [0..z]
   x <- [0..z]
   guard $ 3*x^2 + 2*y + 1 == z
   return (x,y,z)
```

2. »=-Notation

```
solutions'' :: [(Int , Int , Int )]
solutions'' =
  [0..] >>= \z ->
  [0..z] >>= \y ->
  [0..z] >>= \x ->
  (guard $ 3*x^2 + 2*y + 1 == z) >>
  return (x,y,z)
```

Aufgabe 12.2 (2 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.1* Maybe- und Listenmonaden *gelesen werden.*
Definieren Sie die Haskell-Funktion `f` vom Typ `Int -> Int -> Int -> Maybe Int`. Wie in Aufgabe 11.3 soll diese mit den Funktionen `safeDiv` und `safeSqrt` gelöst werden und die Gleichung $f(x,y,z) = \frac{\sqrt{x}}{\sqrt{\frac{y}{z}}}$ erfüllen. Diesmal soll jedoch sonst nur die Monadeneigenschafft von `Maybe` genutzt werden.

```
intsqrt :: Int -> Int
intsqrt = floor . sqrt . fromIntegral

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

safeSqrt :: Int -> Maybe Int
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (intsqrt x)
```

Lösungsvorschlag

```
f :: Int -> Int -> Int -> Maybe Int
f x y z = do
  yz <- safeDiv y z
  sqrt1 <- safeSqrt x
  sqrt2 <- safeSqrt yz
  safeDiv sqrt1 sqrt2
```

Aufgabe 12.3 (4 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.10* Schreibermonaden *gelesen werden.*
Schreiben Sie einen Algorithmus, welche die Operation $2*(3+1)+5$ ausführt und dabei jeden Schritt protokolliert.
Gehen Sie dazu schrittweise vor.

1. Schreiben Sie die Funktionen `addW` und `multW`, welche zwei Ganzzahlen addieren bzw. multiplizieren und den Vorgang protokollieren. Definieren Sie die Funktionen mithilfe der `do`-Notation und der Funktion `tell`. Die Funktion `tell` schreibt einen beliebigen `String` in das Protokoll.
   Vorgaben:

   ```
   type Writer s a = (s,a)

   tell :: s -> Writer s ()
   tell s = (s,())

   addW, multW :: Int -> Int -> Writer String Int
   ```

2. Definieren Sie das Programm `progW` vom Typ `Writer String Int`, welches die Operation $2*(3+1)+5$ ausführt. Nutzen Sie auch hier die `do`-Notation.
   Beispiel: `progW` $\rightsquigarrow$ `("3+1 = 4\n2*4 = 8\n8+5 = 13\n",13)`

Lösungsvorschlag

1. Addition und Multiplikation

   ```
   addW x y = do
     tell $ show x ++ "+" ++ show y ++ "␣=␣" ++ show r ++ "\n"
     return r
     where r = x + y

   multW x y = do
     tell $ show x ++ "*" ++ show y ++ "␣=␣" ++ show r ++ "\n"
     return r
     where r = x * y
   ```

2. $2 * (3 + 1) + 5$

```
progW :: Writer String Int
progW = do
  r1 <- addW 3 1
  r2 <- multW 2 r1
  addW r2 5
```

Aufgabe 12.4 (4 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.9* Lesermonaden *gelesen werden.*
Schreiben Sie die Funktion `bexp2store` aus Aufgabe 7.4 so um, dass sie Gebrauch von der Lesermonade macht. Es gelten folgende Typen:

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

Lösungsvorschlag

```
bexp2store True_ _ = return True
bexp2store False_ _ = return False
bexp2store (BVar x) _ = ($x)
bexp2store (Or bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return (or is)
bexp2store (And bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return (and is)
bexp2store (Not bs) st = do
  i <- bexp2store bs st
  return (not i)
bexp2store (e1 := e2) st
  = return (exp2store e1 st == exp2store e2 st)
bexp2store (e1 :<= e2) st
  = return (exp2store e1 st <= exp2store e2 st)
```

Alternativ kann auch der Stil aus Aufgabe 12.3 verwendet werden.

```
type Reader s a = s -> a

ask :: Reader s s
ask = id

bexp2store' ::  BExp x -> Store x -> Reader (BStore x) Bool
bexp2store' (BVar x) _ = do
  bst <- ask
  return (bst x)
```

Die restlichen Muster (Pattern) von `bexp2store'` seien exakt gleich definiert wie bei `bexp2store`.