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 11

Ausgabe: 12.2.2017, **Abgabe:** 19.2.2017 – 16:00 Uhr, **Block:** 6

Lesen Sie bitte selbstständig die Abschnitte *Lesermonaden* und *Schreibermonaden* auf den Folien 205–208 und den Abschnitt *Zustandsmonaden* auf den Folien 213–215.

Aufgabe 11.1 (4 Punkte) *Lesermonade*
Schreiben Sie die Funktion `bexp2store` so um, dass sie Gebrauch von der Lesermonade macht. Verwenden Sie die do-Notation.

```
type BStore x = x -> Bool

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 b) st bst = not $ bexp2store b st bst
bexp2store (e1 := e2) st _ = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) st _ = exp2store e1 st <= exp2store e2 st
```

Lösungsvorschlag

```
bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store True_ _ = return True
bexp2store False_ _ = return False
bexp2store (BVar x) _ = do
  bst <- id
  return $ bst 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 b) st = do
  i <- bexp2store b 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
```

Aufgabe 11.2 (4 Punkte) *Schreibermonade*
Gegeben sei folgende Vorlage.

```
type ID = Int
type Bank = [(ID,Account)]
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

own1, own2, own3 :: Client
own1 = Client "Max" "Mustermann" "Musterhausen"
own2 = Client "John" "Doe" "Somewhere"
own3 = Client "Erika" "Mustermann" "Musterhausen"

acc1, acc2, acc3 :: Account
acc1 = Account 100 own1
acc2 = Account 0 own2
acc3 = Account 50 own3

bank :: Bank
bank = [(1,acc1), (2,acc2), (3,acc3)]

credit :: Int -> ID -> Bank -> Bank
credit amount id ls
  = updRel ls id entry{ balance = oldBalance + amount} where
    Just entry = lookup id ls
    oldBalance = balance entry

debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)
```

Erweitern Sie die Vorlage um eine Transferfunktion mit Protokollierung (Log). Machen Sie hierfür Gebrauch von einer Schreibermonade mit `String` für die Protokollierung (`(,) String`).

a) Eine Loggerfunktion `logMsg`, welche einen beliebigen `String` an das Protokoll anhängt. Geben Sie auch den Typ der Funktion an.

b) `transferLog :: Int -> ID -> ID -> Bank -> (String,Bank)` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite und schreibt einen Eintrag in das Protokoll. Der Eintrag soll folgendes Format besitzen.

   "Der Betrag *<ammount>* wurde von Konto *<id1>* auf Konto *<id2>* übertragen."

   Verwenden Sie die do-Notation. Der Tupelkonstruktor `(,)` darf *nicht* benutzt werden.

c) `transactions :: Bank -> (String,Bank)` – Soll Transaktionen durchführen, welche zu folgenden Protokolleinträgen passen.

```
putStrLn $ fst $ transactions bank ↪

Der Betrag 50 wurde von Konto 1 auf Konto 2 übertragen.
Der Betrag 25 wurde von Konto 1 auf Konto 3 übertragen.
Der Betrag 25 wurde von Konto 2 auf Konto 3 übertragen.
```

Verwenden Sie die do-Notation. Der Tupelkonstruktor `(,)` darf *nicht* benutzt werden.

Lösungsvorschlag

```
logMsg :: String -> (String,())
logMsg msg = (msg,())

transferLog :: Int -> ID -> ID -> Bank -> (String,Bank)
transferLog amount id1 id2 bank = do
  logMsg "Der Betrag "
  logMsg $ show amount
  logMsg " wurde von Konto "
  logMsg $ show id1
  logMsg " auf Konto "
  logMsg $ show id2
  logMsg " übertragen.\n"
  return $ debit amount id1 $ credit amount id2 bank

transactions :: Bank -> (String,Bank)
transactions bank = do
  bank2 <- transferLog 50 1 2 bank
  bank3 <- transferLog 25 1 3 bank2
  transferLog 25 2 3 bank3
```

Aufgabe 11.3 (4 Punkte) *Zustandsmonade*
Benutzen Sie die Vorlage aus Aufgabe 11.2 und erweitern Sie diese um die folgenden Funktionen.

a) `putAccount :: ID -> Account -> State Bank ()` – Erstellt ein neues Konto mit angegebener `ID`. Ist bereits ein Konto mit der `ID` vorhanden, wird dieses überschrieben.

b) `getAccount :: ID -> State Bank (Maybe Account)` – Falls vorhanden, wird das Konto mit der `ID` ausgegeben.

c) `creditS :: Int -> ID -> State Bank ()` – Addiert den angegebenen Betrag auf das angegebene Konto. Verwenden Sie die do-Notation. Konstruktor und Destruktor von `State` (`State`, `runS`) dürfen *nicht* verwendet werden.

d) `debitS :: Int -> ID -> State Bank ()` – Subtrahiert den angegebenen Betrag von dem angegebenen Konto. Verwenden Sie die do-Notation. Konstruktor und Destruktor von `State` dürfen *nicht* verwendet werden.

e) `transferS :: Int -> ID -> ID -> State Bank ()` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite. Verwenden Sie die `do`-Notation. Konstruktor und Destruktor von `State` dürfen *nicht* verwendet werden.

Beispiel: Mit `transferS 25 1 3` werden 25 Geldeinheiten von Konto 1 auf Konto 3 übertragen.

```
map (fmap balance) $ snd $ runS (transferS 25 1 3) bank
  ⇝[(1,75),(2,0),(3,75)]
```

Lösungsvorschlag

```
putAccount :: ID -> Account -> State Bank ()
putAccount id acc = State $ \bank -> ((), updRel bank id acc)

getAccount :: ID -> State Bank (Maybe Account)
getAccount id = State $ \bank -> (lookup id bank,bank)

creditS  :: Int -> ID -> State Bank ()
creditS amount id = do
  Just entry <- getAccount id
  let oldBalance = balance entry
  putAccount id entry{ balance = oldBalance + amount }

debitS :: Int -> ID -> State Bank ()
debitS amount = creditS (-amount)

transferS :: Int -> ID -> ID -> State Bank ()
transferS amount id1 id2 = do
  debitS amount id1
  creditS amount id2
```