technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 1

Aufgabe 1.1

Installieren Sie die Haskell-Platform (`http://www.haskell.org/platform/`) auf ihrem Rechner. Stellen Sie dabei sicher, dass `ghc` und `ghci` zu ihrer Pfadvariablen hinzugefügt sind.

1. Öffnen Sie den Texteditor Ihrer Wahl und tippen Sie folgendes Programm ab:

   ```
   f :: Int -> Int -> Int -> Int
   f x y z = x + y + z * z
   ```

2. Speichern Sie das Programm in einer Datei mit der Endung `.hs`. Den Pfad zu der Datei nennen wir im Folgenden `file.hs`.

3. Öffnen Sie die Kommandozeile und laden Sie die Datei mit dem interaktiven Modus des GHC (GHCi genannt), wie folgt: `ghci file.hs`
   Sie sollten nun die folgende Ausgabe erhalten:

   ```
   [1 of 1] Compiling Main             ( file.hs, interpreted )
   Ok, modules loaded: Main.
   *Main>
   ```

4. Rufen Sie nun die Funktion `f` auf, indem Sie zum Beispiel `f 1 2 3` eingeben und mit ENTER bestätigen. Das Ergebnis wird ausgegeben und Sie können weitere Funktionsaufrufe auswerten lassen.

Folgende Kommandos des GHCi haben sich als nützlich erwiesen:

- `:load file` (kurz `:l`) läd die Datei `file` in den GHCi.
- `:reload` (kurz `:r`) läd die aktuelle Datei neu ein. Nachdem Änderungen an dem Quelltext vorgenommen wurden, kann die aktuelle Datei mit `:r` leicht neu geladen werden.
- `:type ausdruck` (kurz `:t`) zeigt den Typ des Ausdruckes `ausdruck` an, z.B. `:t f` oder `:t f 1 2 3`.
- `:help` (kurz `:h`) öffnet die Hilfe mit weiteren nützlichen Befehlen.
- `:quit` (kurz `:q`) beendet den GHCi.

Aufgabe 1.2

Die folgende Aufgabe entählt eine Reihe von fehlerhaften Haskell-Ausdrücken. Ziel dieser Aufgabe ist, dass Sie sich mit den Fehlermeldungen des GHCi vertraut machen. Laden Sie dazu die Datei aus Aufgabe 1 und interpretieren Sie die folgenden Ausdrücke mit dem GHCi. Versuchen Sie die Fehlermeldungen nachzuvollziehen.

1. `f 3 1 True`
2. `f 4 3 2 1`
3. `f 3 2 1`
4. `foo 3 2 1`

Aufgabe 1.3

Gegeben sei die Funktion `addFive` und die konstante Funktion `one`:

```
addFive :: Int -> Int
addFive x = x + 5

one :: Int
one = 1
```

1. Definieren Sie eine Konstante `k :: Int`, deren Auswertung die Zahl `11` liefert. Nutzen Sie dabei die Funktionen `addFive` und `one`.
2. Definieren Sie eine Funktion `addTen :: Int -> Int`, die eine ganze Zahl als Parameter erhält und die Summe aus Zehn und der übergebenen Zahl berechnet. Nutzen Sie dafür die Funktion `addFive` und den Dollar-Operator.
3. Definieren Sie eine Funktion `addTenComposition :: Int -> Int`, die semantisch äquivalent zu der Funktion `addTen` ist, jedoch die Funktion `addFive` und die Funktionskomposition nutzt.

Aufgabe 1.4

Werten Sie den folgenden Ausdruck unter Angabe von Zwischenergebnissen aus:

$$(\lambda y.(\lambda x.x + 1)\ (4 + y))\ 6$$

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 2

Aufgabe 2.1 Die folgenden Funktionen in Lambda-Notation sind gegeben.

```
f1 = \x -> x + 1

f2 = \(x,y) -> (x * y + 1)

f3 = \x -> \y -> y (x * x)

f4 = \f g -> f . g . f
```

1. Formen Sie die Funktionen in die applikative Notation um.
2. Formen Sie die Funktionen in die Präfixschreibweise (Folie 15) um, indem Sie alle Funktionsanwendungen in Infixnotation in die Präfixnotation überführen. Aus `x + y` wird dabei beispielsweise `(*) x y`.
3. Bestimmen Sie die Typen der Funktionen ohne Compilerunterstützung wie folgt:
   - Bestimmen Sie die Typen der Funktionen `f1`, `f2` und `f3` formal nach den Regeln auf Folie 14. Die Anwendbarkeit der Regeln wird deutlicher, wenn Sie die Funktionen in Präfixnotation betrachten (siehe Aufgabe 2.1.2).
   - Leiten Sie den Typ von `f4` hingegen informell ab.

   Nehmen Sie dabei jeweils an, dass `1 :: Int` und `(+), (*) :: Int -> Int -> Int` gilt.
4. Entscheiden Sie für die in der vorherigen Aufgabe bestimmten Typen, ob es sich jeweils um einen monomorphen oder polymorphen Typen handelt.

Aufgabe 2.2 Gegeben sei die folgende Funktion `add4Ints`:

```
add4Ints :: Int -> Int -> Int -> Int -> Int
add4Ints v w x y = v + w + x + y
```

1. Die Funktionsanwendung in Haskell ist per Definition bekanntermaßen linksassoziativ. Erweitern Sie den Ausdruck `add4Ints 1 2 3 4` um die implizit vorhandenen Klammern.
2. Die Linksassoziativität der Funktionsanwendung hat direkt Folgen für den Typ von Funktionen mit mehreren Argumenten. Wie sieht die explizite Klammerung für den Typ der Funktion `add4Ints` (also `Int -> Int -> Int -> Int -> Int`) aus?

Hinweis: Die Explikation der Klammerung erhält natürlich die Semantik des Ausdrucks.

Aufgabe 2.3 Die Collatz-Funktion ist wie folgt definiert:

$$c(n) = \begin{cases} n/2, & \text{falls n gerade} \\ 3n+1, & \text{falls n ungerade} \end{cases}$$

1. Implementieren Sie eine Haskell-Funktion `collatz :: Int -> Int`, die für eine natürliche Zahl das nächste Element in der Collatz-Folge berechnet.

   Gehen Sie also davon aus, dass die Funktion `collatz` nur natürliche Zahlen als Argument erhält. Die Funktion `div :: Int -> Int -> Int` implementiert die Integerdivision in Haskell.

2. Die Collatz-Folge ergibt sich aus der mehrfachen Anwendung der Funktion c auf eine natürliche Zahl n. Man nimmt an, dass die Collatz-Folge für jede natürliche Zahl irgendwann einmal den Wert 1 erreicht.

   Implementieren Sie eine Funktion `collatzMax :: Int -> Int`, die für eine gegebene natürliche Zahl n die größte Zahl zurückgibt, die die Collatz-Folge für Startwert n annimmt.

   Beispiele:

   - $n = 4 : (4, 2, 1)$, `collatzMax 4` $\rightsquigarrow$ `4`
   - $n = 21 : (21, 64, 32, 16, 8, 4, 2, 1)$, `collatzMax 21` $\rightsquigarrow$ `64`

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 3

Aufgabe 3.1 Typinferenz

Diese Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `4 :: Int` gilt.

1. `dropWhile (/=4)`
2. `last . ([4]++)`
3. `λx -> (reverse . map (λx -> 4*x)) x`

Aufgabe 3.2 Endrekursion

Implementieren Sie zwei Funktionen `fRec :: Int -> Float` und `fIter :: Int -> Float`, die die folgende Summe rekursiv bzw. iterativ (siehe Folie 22) berechnen:

$$f(n) =_{df} \sum_{1 \leq i \leq n} \frac{1}{i}$$

Hinweis: Sie können die folgende Funktion zur Konvertierung von `Int` nach `Float` nutzen:

```
intToFloat :: Int -> Float
intToFloat = fromIntegral
```

Aufgabe 3.3 Listen

1. Definieren Sie eine Haskell-Funktion `conjunction :: [Bool] -> Bool`, die genau dann den Wert `True` zurückgibt, wenn die übergebene Liste ausschließlich den Wert `True` enthält. Im Fall der leeren Liste soll der Wert `True` zurückgegeben werden.
2. Definieren Sie eine Haskell-Funktion `fromTo :: Int -> Int -> [Int]`, die für zwei Parameter `begin` und `end` die sortierte Liste der ganzen Zahlen zwischen `begin` und `end` erzeugt. Die beiden Grenzen `begin` und `end` sollen in der Liste enthalten sein.

   Beispielsweise führt der Ausdruck `fromTo (-3) 6` zu folgendem Ergebnis:

   `[-3,-2,-1,0,1,2,3,4,5,6]`
3. Definieren Sie eine Haskell-Funktion `squares :: Int -> [Int]`, die für eine positive natürliche Zahl n die ersten n Quadratzahlen berechnet. Gehen Sie davon aus, dass `squares` nur mit natürlichen Zahlen als Parameter aufgerufen wird. Die Funktion `fromTo` könnte bei der Definition von `squares` hilfreich sein.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 4

Aufgabe 4.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass alle Zahlen vom Typ `Int` und die Funktionen `(+),(*)` vom Typ `Int -> Int -> Int` sind.

1. `map (foldr (+) 0)`
2. `any (<100) . map (λx -> (x - 1)*4)`
3. `zipWith ($) [(+),(*)]`

Aufgabe 4.2 Binärkodierung

Wir betrachten in dieser Aufgabe die Überführung eines Bitstrings in die entsprechende Dezimalzahl. Implementieren Sie zwei semantisch äquivalente Funktionen `binaryToDecimalRec` und `binaryToDecimal` nach den folgenden Vorgaben. Beide Funktionen sollen vom Typ `[Bool] -> Int` sein.

1. Definieren Sie die Funktion `binaryToDecimalRec`, indem Sie eine lokale Hilfsfunktion explizit rekursiv über die Struktur der Eingabeliste definieren. Die Hilfsfunktion muss als zusätzliches Argument den aktuellen Exponenten mitschleppen.
2. Definieren Sie die Funktion `binaryToDecimal` implizit rekursiv, indem Sie (unter anderem) die Funktion `zip` oder `zipWith` nutzen.

Aufgabe 4.3 Pythagoreische Tripel

Ein Tripel $(a, b, c) \in \mathbb{N}^3$ ist ein pythagoreisches Tripel genau dann, wenn $a < b < c$ und $a^2+b^2 = c^2$ gelten. Zum Beispiel handelt es sich bei $(3, 4, 5)$ um ein pythagoreisches Tripel.

Implementieren Sie eine Haskell-Funktion `pyTriples :: [(Int,Int,Int)]`, die die unendliche Liste aller pythagoreischen Tripel erzeugt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 5

Gegeben sei der aus der Vorlesung bekannte Datentyp für Listen:

```
data List a
  = Nil
  | Cons a (List a)
  deriving Show
```

Die natürlichen Zahlen $\mathbb{N}$ können induktiv wie folgt gebildert werden:

- Die Zahl `Zero` ist eine natürliche Zahl.
- Ist `n` eine natürliche Zahl, dann ist auch `Succ n` eine natürliche Zahl.

Folgender Haskell-Datentyp repräsentiert also natürliche Zahlen:

```
data Nat = Zero | Succ Nat deriving Show
```

Aufgabe 5.1 Typinferenz Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `5 :: Int` ist.

1. `foldl (λacc x -> acc || (x ‘mod‘ 5 == 0)) False`
2. `Cons (Succ Zero)`
3. `foldr Cons Nil`

Aufgabe 5.2 Listen

Die folgenden Funktionen sollen rekursiv über die Struktur der Argumente implementiert werden.

1. Definieren Sie eine Haskell-Funktion `mapList :: (a -> b) -> List a -> List b`, die die übergebene Funktion auf jedes Listenelement anwendet.
2. Definieren Sie eine Haskell-Funktion `foldrList :: (a -> b -> b) -> b -> List a -> b`, die eine Faltung von rechts auf einer Liste vom Typ `List a` realisiert.
3. Definieren Sie eine Haskell-Funktion `get :: Int -> List a -> Maybe a`, die einen indexbasierten Zugriff auf Listenelemente implementiert. Sorgen Sie mit Hilfe des Datentyps `Maybe a` dafür, dass die Funktion `get` für **alle** möglichen Eingaben einen Wert (vom Typ `Maybe a`) zurückgibt.

Aufgabe 5.3 Natürliche Zahlen und Listen

1. Definieren Sie eine Haskell-Funktion `toInt :: Nat -> Int`, die eine natürliche Zahl in den repräsentierten `Int` überführt.

2. Definieren Sie eine Haskell-Funktion `getNat :: Nat -> List a -> Maybe a` analog zu Aufgabe 5.2.3, die jedoch statt einem Wert vom Typ `Int` einen Wert vom Typ `Nat` als Index erhält. Ein Aufruf der Funktion `get` aus Aufgabe 5.2.3 ist nicht erlaubt.

3. Definieren Sie eine Haskell-Funktion `len :: List a -> Nat`, die die Länge einer Liste bestimmt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 6

Aufgabe 6.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmen Sie die Typen der folgenden Ausdrücke.

1. `zipWith (/=)`
2. `λy -> map (λx -> x == y)`

Aufgabe 6.2 Typklassen `Eq` und `Ord`

Wir betrachten hier erneut die Typen `Nat` und `List a` von Präsenzblatt 5:

```
data Nat = Zero | Succ Nat deriving Show

data List a = Nil | Cons a (List a) deriving Show
```

Eine Instanz von `Eq` muss entweder `(==)` oder `(/=)` implementieren. Eine Instanz von `Ord` muss entweder `(<=)` oder `compare` implementieren.

1. Definieren Sie eine Instanz der Typklasse `Eq` für den Datentyp `Nat`.
2. Definieren Sie eine Instanz der Typklasse `Ord` für den Datentyp `Nat`.
3. Definieren Sie eine Instanz der Typklasse `Eq` für den Datentyp `List a`.

Aufgabe 6.3 Typklasse `Show`

In der Prelude ist die Typklasse `Show` wie folgt (etwas vereinfacht) definiert:

```
class Show a where
  show :: a -> String
```

Die Funktion `show` gibt an, wie ein Wert ausgegeben wird.

1. Definieren Sie eine `Show`-Instanz für `Nat`, indem Sie die Funktion `show` mit folgendem Verhalten implementieren: `show (Succ (Succ Zero))` ⇝ `"2"`
2. Definieren Sie eine `Show`-Instanz für `List a`, indem Sie die Funktion `show` mit folgendem Verhalten implementieren: `show (Cons 3 $ Cons 5 $ Cons 9 Nil)` ⇝ `"3 , 5 , 9"`

Hinweis: Um nicht zwei Instanzen der Typklasse `Show` vorliegen zu haben, müssen Sie vor Bearbeitung der Aufgabe die automatisch erzeugte `Show`-Instanz (`deriving Show`) für die entsprechenden Typen entfernen.

Aufgabe 6.4 Modellierung

1. Modellieren Sie einen Datentyp mit den folgenden Eigenschaften. Definieren Sie dabei geschickt eigene Typen, um die Modellierung besser zu strukturieren.

   - Eine Firma besteht aus mehreren Abteilungen.
   - Jede Abteilung hat einen Abteilungsleiter und mehrere Mitarbeiter.
   - Abteilungsleiter und Mitarbeiter haben einen Namen und ein Wert für das monatliche Gehalt.

   Hinweis: Fügen Sie jeder Datentypdefinition ein `deriving Show` an, um Werte des Datentyps anzeigen zu können.

2. Definieren Sie eine nichttriviale Beispielinstanz des Datentyps.

3. Implementieren Sie eine Funktion, die eine Firma als Argument erhält und das Gehalt aller Mitarbeiter verdoppelt.

4. Implementieren Sie eine Funktion, die eine Firma als Argument erhält und die Namen aller Abteilungsleiter zurückgibt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 7

Aufgabe 7.1 Typklasse `Monoid`

Die Typklasse `Monoid` aus dem Modul aus der Standardbibliothek ist leicht vereinfacht wie folgt definiert:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

1. Definieren Sie für den Datentyp `data Nat = Zero | Succ Nat` eine `Monoid`-Instanz, so dass `mempty` die Null und `mappend` die Addition zweier natürlicher Zahlen darstellt.

2. Gegeben sei der Datentyp `Endo a` zur Repräsentation von Endofunktionen:

   ```
   data Endo a = Endo { getEndo :: a -> a }
   ```

   Definieren Sie eine `Monoid`-Instanz für `Endo a` unter der Funktionskomposition.

3. Die Instanzen von `Monoid` sollten folgende drei Regeln erfüllen:

   - `mapppend mempty x = x`
   - `mappend x mempty = x`
   - `mappend x (mappend y z) = mappend (mappend x y) z`

   Beweisen Sie durch Programmtransformationen, dass Ihre Implementierung für `Endo a` diese Regeln befolgt. Sie können dabei davon ausgehen, dass die Funktionskomposition assoziativ ist.

Aufgabe 7.2 `Collections`

1. Definieren Sie einen Datentyp `Collection a`, der als Attribute eine Liste `[a]` und ein Prädikat `a -> [a] -> Bool` hat.

   Die Semantik des Datentyps ist, dass nur Werte `x` in die Liste `xs` aufgenommen werden, wenn das Prädikat `p` für `x` und `xs` als Argumente `True` liefert.

2. Geben Sie eine Instanz der Typklasse `Show` für `Collection a` an.

3. Definieren Sie eine Funktion `empty :: (a -> [a] -> Bool) -> Collection a`, die eine leere `Collection` mit gegebenem Prädikat erzeugt.

4. Definieren Sie eine Funktion `insert :: a -> Collection a -> Collection a`, die einen Wert in die `Collection` aufnimmt, wenn das Prädikat für die aktuelle Liste und den einzufügenden Wert `True` liefert.

5. Definieren Sie als Beispiel für eine `Collection` den Wert `set :: Eq a => Collection a`, der eine Menge implementiert.

   Beispiel: `insert 3 $ insert 1 $ insert 1 set` $\rightsquigarrow$ `[3,1]`

6. Definieren Sie als weiteres Beispiel eine Collection `palindromes :: Eq a => Collection [a]`, die nur Palindrome aufnimmt. Ein Palindrom ist ein Wort, das von vorne und hinten gelesen dasselbe ergibt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 8

Aufgabe 8.1 Typklasse `Graphen`

Der Datentyp `Tree v e` repräsentiert Bäume mit Knoten- und Kantenmarkierung.

```
data Tree v e = Node { node :: v, children :: [(e,Tree v e)] }
```

Aus dem aus der Vorlesung bekannten Typ `Graph` erhält man durch entsprechende Festlegung der Typvariablen den Typ `Graph a (a,label)` für Graphen mit Knoten- und Kantenmarkierungen:

```
type NodeAndEdgeLabeled v e = Graph v (v,e)
```

Implementieren Sie eine Haskell-Funktion zur Überführung eines Baumes in Termdarstellung in die Graphdarstellung:

```
treeToGraph :: Eq v => Tree v e -> NodeAndEdgeLabeled v e
```

Gehen Sie davon aus, dass die Knotenmarkierungen in dem Baum eindeutig sind und sie daher als jeweils eindeutige Identifikatoren dienen.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 9

Aufgabe 9.1 Kinds

Gegeben seien folgende Datentypen:

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data WrapInt f = WrapInt (f Int)
data T f g = T (f Bool) (g Int Bool)
```

1. Bestimmen Sie den Kind der folgenden Typen bzw. Typkonstruktoren:

   `Bool`, `Maybe`, `Either`, `WrapInt`, `T`.

2. Geben Sie für die Typen `WrapInt f` und `T f g` je zwei beliebige Werte an.

Aufgabe 9.2 Funktoren

Wir betrachten die aus der Prelude bekannte Typklasse `Functor`:

```
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

Zudem sei folgender Datentyp für Binärbäume gegeben:

```
data Bintree a = Leaf | Branch a (Bintree a) (Bintree a)
```

1. Instanziieren Sie die Typklasse `Functor` für den Typ `Bintree`.

2. Beweisen Sie, dass Ihre Definition aus Aufgabenteil 1 die beiden Regeln für Funktoren erfüllt. Jede Instanz der Typklasse `Functor` muss folgende Regeln erfüllen:

   ```
   fmap id = id
   fmap (f . g) = fmap f . fmap g
   ```

   Alternativ kann man auch sagen, dass für alle `x :: Bintree a` folgendes gelten muss:

   ```
   fmap id x = x
   fmap (f . g) x = fmap f (fmap g x)
   ```

   Hinweis: Nutzen Sie eine vollständige Induktion über die Tiefe des Baumes als Beweistechnik.

3. Definieren Sie eine Funktion `doubleValues :: Functor f => f Int -> f Int`, die die Funktion `fmap` nutzt, um alle Werte in einem beliebigen Funktor zu verdoppeln. Überprüfen Sie die Funktionsweise der Funktion `doubleValues` an den Funktoren `Bintree`, `[]` und `Maybe`.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 10

Aufgabe 10.1 Partielle Funktionen

Gegeben seien folgende drei partielle Funktionen:

```
logarithmus :: Float -> Maybe Float
logarithmus x
  | x <= 0    = Nothing
  | otherwise = Just (log x)

quadratwurzel :: Float -> Maybe Float
quadratwurzel x
  | x < 0     = Nothing
  | otherwise = Just (sqrt x)

kehrwert :: Float -> Maybe Float
kehrwert x
  | x == 0 = Nothing
  | x /= 0 = Just (1 / x)
```

Implementieren Sie partielle Funktionen `fCase, fBind, fDo, fKleisli :: Float -> Maybe Float`, die jeweils den Kehrwert der Wurzel des Logarithmus' ihres Arguments berechnen. Formulieren Sie die vier semantisch äquivalenten Funktionen wie folgt:

1. Benutzen Sie die `case`-Syntax zur Fehlerbehandlung in der Funktion `fCase`.
2. Benutzen Sie die »=-Notation zur Fehlerbehandlung in der Funktion `fBind`.
3. Benutzen Sie die `do`-Notation zur Fehlerbehandlung in der Funktion `fDo`.
4. Benutzen Sie die Kleisli-Komposition in der Funktion `fKleisli`.

Aufgabe 10.2 `Listenmonade`

Reduzieren Sie folgenden Ausdruck, indem Sie

1. die Listenkomprehension in die `do`-Notation überführen,
2. dann in die »=-Notation wechseln
3. und die Definition von »= einsetzen.

```
xs :: [(Int,Int)]
xs = [ (x,y) | x <- [1,2] , y <- [6,7] , x + y /= 8]
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 11

Aufgabe 11.1 Markieren von Bäumen

In dieser Aufgabe ist eine Funktion `numbering :: Int -> Bintree a -> Bintree Int` zu definieren, die die k Knoten eines Baumes beginnend bei einem Wert n mit den Werten von n bis $n + k - 1$ markiert. Die Knotenmarkierungen vom Typ `a` werden dabei verworfen, wobei die Struktur des Baumes aber erhalten bleiben soll.

Der Datentyp `Bintree a` ist dabei wie folgt definiert:

```
data Bintree a = Leaf a | Branch a (Bintree a) (Bintree a)
```

In dieser Aufgabe soll die Lösung dieser Aufgabe mit der Zustandstransitionsmonade geschehen. Sie finden im EWS eine Datei `Blatt11Vorlage.hs`, die Ihnen als Vorlage dienen kann. In dem Modul finden Sie zudem eine Lösung der Aufgabe, die ohne die Transitionsmonade auskommt.

1. Zunächst wird eine Hilfsfunktion `fresh :: Trans Int Int` benötigt, die den aktuellen Zustand als Resultat zurückgibt und den Zustand inkrementiert. Definieren Sie die Funktion `fresh`, indem Sie eine Zustandstransition mit dem Konstruktor `T` erzeugen.

   Beispiele:

   ```
   runT fresh 3 ⇝ (3,4)
   runT fresh 10 ⇝ (10,11)
   ```

2. Definieren Sie nun die Funktion

   ```
   numberTree :: Tree a -> Trans Int (Bintree Int)
   ```

   zur Markierung eines Baumes als Zustandstransition, wobei in dem Zustand stets die als nächstes zu vergebene Markierung steht. Vergeben Sie die Markierungen in der Preorder-Reihenfolge.

   Natürlich sollten sie in dieser Funktion die Hilfsfunktion `fresh` aus dem ersten Aufgabenteil nutzen. Definieren Sie diese Funktionen in »=- oder `do`-Notation.

   Sie können die Funktion `numberT` aus der Vorlage zur Überprüfung Ihrer Implementierung nutzen. Für alle `t :: Bintree a` und `n :: Int` sollte folgende Bedingung gelten:

   ```
   runT (numberTree t) n == numberT t n
   ```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 12

Aufgabe 12.1 Funktionen mit polymorphen und monadischen Rückgabetypen
Implementieren Sie eine Funktion

```
filt :: MonadPlus m => (a -> Bool) -> [a] -> m a
```

die die Funktion `filter` aus der *Prelude* verallgemeinert. Die Funktion `filt` soll sich bei einer Festlegung des Rückgabetyps auf eine Liste von Werten wie `filter` verhalten:

```
filt (<1) [1,-2,3,-4,5] :: [Int] ⇝ [-2,-4]
```

Darüber hinaus soll `filt` aber beispielsweise auch für einen in `Maybe` eingebetteten Wert funktionieren und dann den ersten Wert in der Liste zurückgeben, der das Prädikat erfüllt:

```
filt (<1) [1,-2,3,-4,5] :: Maybe Int ⇝ Just (-2)
```

Gibt es keinen Wert, für den das Prädikat gilt, gibt die Funktion für den Rückgabetyp `Maybe Int` den Fehlerwert `Nothing` zurück.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Jens Lechner (*jens.lechner@tu-dortmund.de*)

Wintersemester 2015/2016

Übungen zu Funktionaler Programmierung
Präsenzblatt 13

Aufgabe 13.1 Dynamische Programmierung mit Feldern

Gegeben sei folgende rekursive Berechnung des i-ten Gliedes der Catalan-Folge:

```
catalan :: Int -> Int
catalan 0 = 1
catalan n = sum (map (\i -> catalan i * catalan (n-1-i)) [0..n-1])
```

Mit Hilfe von dynamischer Programmierung soll nun ein Feld `catalanArr :: Array Int Int` mit den Gliedern der Catalan-Folge definiert werden. Bei der Berechnung der Glieder sollen Zwischenergebnisse durch Zugriffe auf das Feld anstelle durch Rekursion berechnet werden.

```
catalanDyn :: Int -> Int
catalanDyn n = catalanArr ! n where

        catalanArr :: Array Int Int
        catalanArr = mkArray ...
```

Die Funktion `catalanDyn :: Int -> Int` sollte durch die Nutzung der dynamischen Programmierung wesentlich effizienter als die rekursive Variante `catalan` sein.