Funktionale Programmierung

Pascal Hof

10.1.2014

Kinds

Kinds

- Bisher: primitive Typen (Typen erster Ordnung):
  `Int, Bool, Maybe Int, [Int -> Bool] :: *`
- Typen zweiter Ordnung sind Typen, die als Argument genau einen Typen bekommen: `Maybe, [] :: * -> *`.
- Typen höherer Ordnung heißen auch **Typkonstruktoren**.
- Kinds dienen zur Klassifikation von Typen.

Drei Ebenen

Jeder **Wert** hat einen **Typ**.
Jeder **Typ** hat einen **Kind**.
$\rightarrow$ Kinds sind also Typen von Typen.

Kinds

Partielle Anwendung von Typkonstruktoren

```haskell
data Pair a b = Pair a b
```

`Pair :: * -> * -> *`

Typkonstruktoren können auch partiell angewandt werden:

```haskell
Pair Int :: * → *
```

Sonderfall []

Der Typkonstruktor `[] :: * -> *` wird in der Regel wie folgt angewandt:

`[Int] :: *`

Es kann aber auch die Präfix-Notation genutzt werden:

`[] Int :: *`

Kinds

Kindbestimmung

```haskell
data T f = T (f Bool)
```

Der folgende Typ bekommt einen Typkonstruktor als Argument:

```haskell
T :: (* -> *) -> *
```

Bestimmung von Kinds im GHCi

```haskell
> :k Int
Int :: *
> :k Maybe
Maybe :: * -> *
> :k []
[] :: * -> *
> :k [] Bool
[] Bool :: *
```

Funktoren

Typklasse Functor

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

Typen, die die Typklasse `Functor` implementieren, müssen eine freie, primitive Typvariable und somit den Kind `* -> *` haben.

Beispiele für Funktoren

Maybe-Funktor

```haskell
instance Functor Maybe where
  -- fmap :: (a → b) → Maybe a → Maybe b
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Beispiel: Maybe-Funktor

```haskell
fmap (+1) Nothing  ⟹ Nothing
fmap (+1) (Just 2) ⟹ Just 3
```

Beispiele für Funktoren

[]-Funktor

```
instance Functor [] where
  -- fmap :: (a → b) → [a] → [b]
  fmap = map
```

Beispiel: []-Funktor

```
fmap (+1) []      ⟹ []
fmap (+1) [1,2,3] ⟹ [2,3,4]
```

Beispiele für Funktoren

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

```haskell
instance Functor Bintree where
  -- fmap :: (a → b) → Bintree a → Bintree b
  fmap f (Leaf x)          = Leaf (f x)
  fmap f (Branch x t1 t2) =
    Branch (f x) (fmap f t1) (fmap f t2)
```

Beispiel: Bintree-Funktor

```haskell
fmap (+1) (Branch 5 (Leaf 2) (Branch 7 (Leaf 1) (Leaf 3)))
      ⟹ Branch 6 (Leaf 3) (Branch 8 (Leaf 2) (Leaf 4)))
```

Beispiele für Funktoren

```haskell
data BExp a = Var a
            | TTrue
            | Not (BExp a)
            | Conj (BExp a) (BExp a)
```

```haskell
instance Functor BExp where
  -- fmap :: (a → b) → BExp a → BExp b
  fmap f (Var v)       = Var (f v)
  fmap _ TTrue         = TTrue
  fmap f (Not b)       = Not $ fmap f b
  fmap f (Conj b1 b2) = Conj (fmap f b1) (fmap f b2)
```

Beispiel: BExp-Funktor

```haskell
fmap (+1) (Conj (Var 2) (Not (Var 5)))⟹Conj (Var 3) (Not (Var 6))
```

Beispiele für Funktoren

Term-Funktor

```haskell
data Term f v = F f [Term f v] | V v

instance Functor (Term f) where
  -- fmap :: (a -> b) -> Term f a -> Term f b
  fmap h (F f ts) = F f $ map (fmap h) ts
  fmap h (V x)    = V $ h x
```

Gesetze für Funktoren

Nicht jede Instanz der Typklasse `Functor` ist auch wirklich ein Funktor. Bei der Definition einer Instanz der Typklasse `Functor` sollte darauf geachtet werden, dass folgende Gesetze gelten für alle `x` gelten:

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

Der Compiler überprüft die Gesetze nicht automatisch!

Gesetze für Funktoren

Beispiel für Maybe-Funktor

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Beweis des 1.Funktorgesetzes für den Maybe-Funktor

Zu zeigen: `fmap id x = x`. Fallunterscheidung über Struktur von `x`

- 1.Fall: `x = Nothing`

```
fmap id Nothing  = Nothing
```

- 2.Fall: `x = Just y`

```
fmap id (Just y) = Just (id y) = Just y
```

Zusammenfassung

Funktoren

Funktoren müssen vom Kind `* -> *` sein und die folgenden Gesetze erfüllen:

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

Luft holen

Wir betrachten im Folgenden eine Reihe von auf den ersten Blick unabhängigen, kurzen Beispielfunktionen, denen ein gemeinsames Prinzip zugrunde liegt.

Funktionen

nicht abfangbare Laufzeitfehler

```
Prelude> 4 `div` 0
*** Exception: divide by zero
```

```
Prelude> [1,2,3,4] !! 8
*** Exception: Prelude.(!!): index too large
```

Explizite partielle Funktionen

```
data Maybe a = Nothing | Just a
```

Unsichere Funktionen können mit `Maybe` "sicher" gemacht werden.

Sichere Varianten mit Einbettung in Maybe

Sichere Division

```
sdiv :: Int → Int → Maybe Int
sdiv x y
   | y == 0 = Nothing
   | y /= 0 = Just (div x y)
```

Sicherer indexbasierter Zugriff auf Listenelemente

```
sget :: Int → [a] → Maybe a
sget _ []      = Nothing
sget 0 (x:_)   = Just x
sget n (x:xs)
   | n < 0 = Nothing
   | n ≥ 0 = sget (n-1) xs
```

Kombination von partiellen Funktionen

Erinnerung

```
sget :: Int → [a] → Maybe a
```

Indexbasierter Zugriff auf die Liste und Addition der Werte

```
f :: Int  → Int → [Int] → Maybe Int
f i j xs = case sget i xs of
  Nothing → Nothing
  Just x  → case sget j xs of
    Nothing → Nothing
    Just y  → Just $ x + y
```

Das ist offensichtlich nicht schön!

Kombinator zur Komposition von partiellen Funktionen

```haskell
compose :: Maybe a → (a → Maybe b) → Maybe b
compose Nothing  _ = Nothing
compose (Just x) f = f x
```

Völlig äquivalent:

```haskell
compose :: Maybe a → (a → Maybe b) → Maybe b
compose mb f = case mb of
  Nothing → Nothing
  Just x  → f x
```

Erinnerung: Definition von compose

```
compose :: Maybe a → (a → Maybe b) → Maybe b
compose Nothing  _ = Nothing
compose (Just x) f = f x
```

Beispiele zur Nutzung von compose

```
Nothing `compose` λx → Just (x∗10) ⟹ Nothing
```

```
Just 3 `compose` (λx → Just (x∗10) `compose` λy → Just (y+1))
⟹ Just 31
```

```haskell
compose :: Maybe a → (a → Maybe b) → Maybe b
compose mb f = case mb of
  Nothing → Nothing
  Just x  → f x
```

Ausgangsituation:

```haskell
f :: Int  → Int → [Int] → Maybe Int
f i j xs = case sget i xs of
  Nothing → Nothing
  Just x  → case sget j xs of
    Nothing → Nothing
    Just y  → Just (x + y)
```

Inneres case-Statement durch `compose` ersetzen.

```haskell
f :: Int  → Int → [Int] → Maybe Int
f i j xs = case sget i xs of
  Nothing → Nothing
  Just x  → sget j xs `compose` (λy → Just (x + y))
```

```
compose :: Maybe a → (a → Maybe b) → Maybe b
compose mb f = case mb of
  Nothing → Nothing
  Just x  → f x
```

vereinfachte Definition der vorherigen Folie:

```
f :: Int  → Int → [Int] → Maybe Int
f i j xs = case sget i xs of
  Nothing → Nothing
  Just x  → sget j xs `compose` (λy → Just (x + y))
```

Wiederum case-Statement durch compose ersetzen.

```
f :: Int  → Int → [Int] → Maybe Int
f i j xs =
  sget i xs `compose` (λx → sget j xs `compose` (λy → Just (x + y)))
```

Fazit: `Maybe`

Keine lästige, manuelle Fehlerbehandlung mehr nötig, da die Funktion `compose` die Fehlerbehandlung kapselt.

Listen

Gegeben:

- Liste von Werten `xs :: [a]`
- Funktion `f :: a -> [b]`

Gesucht: Funktion `applyAndCollect`, die `f` auf jedes Listenelement von `xs` anwendet und die Ergebnisse in einer neuen Liste `[b]` aufsammelt.

Beispiele

```
[1,5,6] `applyAndCollect` λx → [x*10]  ⟹ [1,50,60]
```

```
[1,5,6] `applyAndCollect` λx → [-x,x]  ⟹ [-1,1,-5,5,-6,6]
```

```
[3,7] `applyAndCollect`
   (λx → [x+2,x+3,x+5] `applyAndCollect` λy → [-y,y])
⟹[-5,5,-6,6,-8,8,-9,9,-10,10,-12,12]
```

Implementierung von `applyAndCollect`

Ein wenig Typinferrenz

`applyAndCollect ::  [a] -> (a -> [b]) -> [b]`

Implementierung von `applyAndCollect`

```haskell
applyAndCollect :: [a] -> (a -> [b]) -> [b]
applyAndCollect xs f = concat (map f xs)
```

Das Ganze in kurz und punktfrei

```haskell
applyAndCollect :: [a] -> (a -> [b]) -> [b]
applyAndCollect = flip concatMap
```

Mit der Funktion `applyAndCollect` können wir Funktionen komponieren, die mehrere Werte zurückgeben.

Variablensubstitution

Ein Datentyp für boolesche Audrücke

```
data BExp a = Var a
            | TTrue
            | Not (BExp a)
            | Conj (BExp a) (BExp a)
```

Variablensubstitution

```
sub :: BExp a → (a → BExp b) → BExp b
sub TTrue          _ = TTrue
sub (Var v)        f = f v
sub (Not b)        f = Not $ sub b f
sub (Conj b1 b2) f = Conj (sub b1 f) (sub b2 f)
```

Variablensubstitution

Erinnerung: Variablensubstitution

```
sub :: BExp a → (a → BExp b) → BExp b
sub TTrue          _ = TTrue
sub (Var v)        f = f v
sub (Not b)        f = Not $ sub b f
sub (Conj b1 b2) f = Conj (sub b1 f) (sub b2 f)
```

Beispiele

```
Conj (Var 'a') (Var 'b') `sub` λv → Var v
⟹ Conj (Var 'a') (Var 'b')
```

```
Conj (Var 'a') (Var 'b') `sub` λv → Not (Var v)
⟹ Conj (Not (Var 'a')) (Not (Var 'b'))
```

```
Conj (Var 'a') (Var 'b') `sub` λ_ → Var 4
⟹ Conj (Var 4) (Var 4)
```

Luft holen

Wir haben jetzt alle Beispiele gesehen.

Typanalyse der definierten Funktionen

```
compose         :: Maybe a → (a → Maybe b) → Maybe b
applyAndCollect :: [a]     → (a → [b]    ) → [b]
sub             :: BExp a  → (a → BExp b ) → BExp b
```

Liste in Präfix-Notation verwenden

```
compose         :: Maybe a → (a → Maybe b) → Maybe b
applyAndCollect :: [] a    → (a → [] b   ) → [] b
sub             :: BExp a  → (a → BExp b ) → BExp b
```

Abstraktion über Typkonstruktoren Maybe, [] und BExp

```
bind :: m a → (a → m b) → m b
```

Erinnerung

```
bind :: m a → (a → m b) → m b
```

Frage

Wie bekomme ich im Allgemeinen einen Wert vom Typ `a` in einen Wert vom Typ `m a` eingebettet?

Blick auf die Beispiele

```
Nothing `compose` λx → Just (x*10)
[1,5,6] `applyAndCollect` λx → [x*10]
Conj (Var 'a') (Var 'b') `sub` λv → Var v
```

Einbettungen

In den Beispielen existieren folgende Funktionen, die einen Wert vom Typ `a` in den jeweiligen Typ `m a` einbetten:

```
Just        :: a → Maybe a
λx → [x]    :: a → [a]
Var         :: a → BExp a
```

Abstraktion über Typkonstruktoren

```
return :: a → m a
```

gemeinsame Schnittstelle

```
class Monad m where
  return :: a → m a
  bind   :: m a → (a → m b) → m b
```

Definition im GHC

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b

  (>>)   :: m a → m b → m b
  f >> g  = f >>= λ_ → g
```

Instanzen von Monad

```
instance Monad Maybe where
  -- return :: a → Maybe a
  return x = Just x

  -- (>>=) :: Maybe a → (a → Maybe b) → Maybe b
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
```

Instanzen von Monad

```haskell
instance Monad [] where
  -- return :: a → [a]
  return x = [x]

  -- (>>=) :: [a] → (a → [b]) → [b]
  (>>=) = flip concatMap
```

Instanzen von Monad

```haskell
instance Monad BExp where
  -- return :: a → BExp a
  return v = Var v

  -- (>>=) :: BExp a → (a → BExp b) → BExp b
  TTrue      >>= _ = TTrue
  Var v      >>= f = f v
  Not b      >>= f = Not (b >>= f)
  Conj b1 b2 >>= f = Conj (b1 >>= f) (b2 >>= f)
```

Instanzen von Monad

Ein einfacher Datentyp

```haskell
data Id a = Id a
```

Die Functor-Instanz

```haskell
instance Functor Id where
  -- fmap :: (a -> b) -> Id a -> Id b
  fmap f (Id x) = Id (f x)
```

Die Monad-Instanz

```haskell
instance Monad Id where
  -- return :: a -> Id a
  return x = Id x
  -- (>>=) :: Id a -> (a -> Id b) -> Id b
  Id x >>= f = f x
```

Die drei Monadengesetze

Wie bei `Functor` sollten Instanzen von `Monad` Gesetze erfüllen:

1. `return x >>= f = f x`
2. `m >>= return = m`
3. `f >>= (g >>= h) = (f >>= g) >>= h`

do-Notation

Syntax

Monaden sind ein so wichtiges Konzept von Haskell, dass es eigenen syntaktischen Zucker für Monaden gibt.

syntaktische Regel

```
m >>= λx → f x      =    do x ← m ; f x
```

do-Notation

Folgende drei Schreibweisen sind äquivalent:

- >>=-Notation

```
m >>= λx → f x
```

- einzeilige do-Notation

```
do x ← m ; f x
```

- mehrzeilige do-Notation

```
do x ← m
   f x
```

do-Notation

Erinnerung: syntaktische Regel

```
m >>= λx → f x      =   do x ← m ; f x
```

Beispiel

```
m = Just 3 >>= λx →       sdiv 37 x >>= λy → Just (y+1)
  = do x   ←  Just 3 ; sdiv 37 x >>= λy → Just (y+1)
  = do x   ←  Just 3 ; y ← sdiv 37 x   ; Just (y+1)
  = do
       x ← Just 3
       y ← sdiv 37 x
       Just (y+1)
```

do-Notation

Folgende drei Schreibweisen für das Beispiel sind äquivalent:

- >>=-Notation

```
m = Just 3 >>= λx → sdiv 37 x >>= λy → Just (y+1)
```

- einzeilige do-Notation

```
m = do x ← Just 3 ; y ← sdiv 37 x ; Just (y+1)
```

- mehrzeilige do-Notation

```
m = do
  x ← Just 3
  y ← sdiv 37 x
  Just (y+1)
```

Einige Monadenkombinatoren

```
when :: Monad m ⇒ Bool → m () → m ()
when b m = if b then m else return ()

sequence :: Monad m ⇒ [m a] → m [a]
sequence []     = return []
sequence (m:ms) = do
  a ← m
  as ← sequence ms
  return $ a:as

sequence_ :: Monad m ⇒ [m a] → m ()
sequence_ = foldr (>>) (return ())

mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM f = sequence ∘ map f
```

Typklasse MonadPlus

Typklasse MonadPlus

```haskell
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Typklasse `MonadPlus` befindet sich im Modul `Control.Monad`.

Gesetze für mzero

1. `mzero >>= f = mzero`
2. `m >>= λx -> mzero = mzero`

Gesetze für mplus

1. ``mzero `mplus` m = m``
2. ``m `mplus` mzero = m``

[]

```haskell
instance MonadPlus [] where
  -- mzero :: [a]
  mzero = []

  -- mplus :: [a] -> [a] -> [a]
  mplus = (++)
```

Beispiele

```haskell
[] `mplus` [1,2,3] `mplus` [5,67]  ==> [1,2,3,5,67]
```

Maybe

```haskell
instance MonadPlus Maybe where
  -- mzero :: Maybe a
  mzero = Nothing

  -- mplus :: Maybe a → Maybe a → Maybe a
  mplus Nothing m = m
  mplus m       _ = m
```

Beispiele

```haskell
Just 3 `mplus` Just 2 `mplus` Just 1  ⟹ Just 3
```

```haskell
Nothing `mplus` Just 2 `mplus` Just 1 ⟹ Just 2
```

```haskell
Just 3 `mplus` Nothing `mplus` Just 1 ⟹ Just 3
```

Der Kombinator guard

Definitionen

```haskell
data () = () -- einelementiger Typ, kein Informationstraeger
```

```haskell
guard :: MonadPlus m ⇒ Bool → m ()
guard b = if b then return () else mzero
```

Verwertung des Ergebnisses bei guard

```haskell
guard b >>= f            -- f :: () → m b
```

Da () keine Information trägt, ignorieren wir die Rückgabe.

```haskell
guard b >>= λ_ → f'    -- f' :: m b
```

Syntaktischer Zucker: `m >> g = m >>= λ_ -> g`

```haskell
guard b >> f'
```

Der Kombinator guard

Erinnerung: Definition von guard

```
guard :: MonadPlus m ⇒ Bool → m ()
guard b = if b then return () else mzero
```

Aufruf von guard mit False

```
  guard False >> m          -- Definition von guard
= mzero       >> m          -- Definition von >>
= mzero      >>= λ_ → m     -- 1. mzero-Gesetz:
= mzero
```

Der Kombinator guard

Erinnerung: Definition von guard

```
guard :: MonadPlus m ⇒ Bool → m ()
guard b = if b then return () else mzero
```

Aufruf von guard mit True

```
  guard True    >>   m            -- Definition von guard
= return ()     >>   m            -- Definition von >>
= return ()    >>=  λ_ → m        -- 1.Monadengesetz
= (λ_ → m) ()
= m
```

Der Kombinator guard

Erinnerung

```haskell
guard :: MonadPlus m ⇒ Bool → m ()
guard b = if b then return () else mzero
```

Sicheres div mit MonadPlus

```haskell
monadPlusSafeDiv :: MonadPlus m ⇒ Int → Int → m Int
monadPlusSafeDiv x y = guard (y /= 0) >> return (x `div` y)
```

Beim Aufruf muss die gewünschte Instanz von MonadPlus durch explizite Typisierung angegeben werden oder diese muss aus dem Kontext ableitbar sein.

Zur Übung: Beweis, dass monadPlusSafeDiv für Maybe der Funktion sdiv entspricht.

Listenkomprehension mit MonadPlus

Jede Listenkomprehension wird intern in eine Folge von monadischen Operationen in der []-Monade übersetzt.

Beispiel

```
xsComp :: [(Int,Int)]
xsComp = [ (x,y) | x ← [1,2,3]
                 , y ← [6,7,8]
                 , x + y /= 8 ]
```

```
xsMPlus :: [(Int,Int)]
xsMPlus = do
  x ← [1,2,3]
  y ← [6,7,8]
  guard $ x + y /= 8
  return (x,y)
```

Zur Übung: `xsMPlus` in `(>>=)`-Schreibweise überführen und reduzieren.

Zusammenfassung

- Kinds als Typen von Typen
- Funktoren als Typklasse mit Funktion `fmap`
- Monaden sind auch nur eine Typklasse mit syntaktischem Zucker bei der Benutzung (`do`-Notation)
- Listenkromprehensionen sind MonadPlus bei der Arbeit