Monaden in Haskell

Joachim Breitner

16. November 2010

Haskell vs. Kategorientheorie

Eine Monade in Haskell ist eine Typklasse mit zwei Methoden:

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

wobei die folgenden Beziehungen gelten sollten:

$$\begin{aligned} \mathsf{return}\ x \gg\!= f &\equiv f\ x \\ a \gg\!= \mathsf{return} &\equiv a \\ (a \gg\!= f) \gg\!= g &\equiv a \gg\!= (\lambda x \to f\ x \gg\!= g). \end{aligned}$$

Eine Formulierung der üblichen Monaden-Definition für Haskell wäre

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

wobei m Typen auf Typen abbildet, also zusammen mit map einen Funktor **Hask** $\to$ **Hask** definiert. Es entspricht weiter return dem ε und join dem μ.

Aus den Bedingungen dass ε und μ natürliche Transformationen sind, sowie den zwei definierenden kommutativen Diagrammen erhalten wir in Haskell-Schreibweise folgende Eigenschaften:

$$\begin{aligned} \mathsf{map}\ g \circ \mathsf{return} &\equiv \mathsf{return} \circ g \\ \mathsf{map}\ g \circ \mathsf{join} &\equiv \mathsf{join} \circ \mathsf{map}\ (\mathsf{map}\ g) \\ \mathsf{join} \circ \mathsf{map}\ \mathsf{join} &\equiv \mathsf{join} \circ \mathsf{join} \\ \mathsf{join} \circ \mathsf{return} &\equiv \mathsf{join} \circ \mathsf{map}\ \mathsf{return} \equiv \mathbf{id}. \end{aligned}$$

Die beiden Definitionen sind äquivalent. Man kann

```
map f = (λa → a >>= (return ∘ f))
join a = a >>= id
```

bzw. `a >>= f = join (map f a)`

definieren und nachrechnen, dass die obigen Bedingungen ineinander übergehen.

Quellen: http://www.haskell.org/haskellwiki/Category_theory/Monads
http://www.haskell.org/haskellwiki/User:Michiexile/MATH198
http://www.haskell.org/all_about_monads/
http://comonad.com/reader/

Ein kleiner Monadenzoo

Im folgenden die bekanntesten Haskell-Monaden, in etwas liberalerer Syntax.

Maybe
(Modelliert: partielle Funktionen, Berechnungen mit Fehlerzuständen)

```
data Maybe a = Just a | Nothing
instance Monad Maybe where
    return = Just
    (Just x) >>= k = k x
    Nothing >>= k = Nothing
```

Either e
(Modelliert: Berechnungen mit Fehlermeldungen)

```
data Either e a = Left e | Right a
instance Monad (Either e) where
    return = Right
    (Left e) >>= k = Left e
    (Right x) >>= k = k x
```

[]
(Liste. Modelliert: Nichtdeterminismus)

```
data [a] = a:[a] | []
instance Monad [] where
    return x = [x]
    [] >>= k = []
    (x:xs) >>= k = k x ++ (xs >>= k)
```

Reader r
(Leser-Monade. Modelliert: Kontextinformationen)

```
type Reader r a = r → a
instance Monad (Reader r) where
    return x = λr → x
    m >>= k = λr → k (m r) r
```

Writer m
(Schreiber-Monade. Modelliert: Protokollierung. (M,e,∗) muss ein Monoid sein.)

```
type Writer m a = (m,a)
instance Monoid m ⇒ Monad Writer where
    return x = (e, x)
    m >>= k = let (m, y) = m
                  (m', z) = k a
              in (m ∗ m', z)
```

State s
(Zustands-Monade. Modelliert: Berechnungen mit Zustand)

```
type State s a = s → (a,s)
instance State s where
    return x = λs → (x,s)
    m >>= k = λs → let (y, s') = m s
                   in k y s'
```

Cont r
(Fortführungs-Monade. Modelliert Sprünge im Programmablauf)

```
type Cont r a = (a → r) → r
instance Monad (Cont r) where
    return x = λc → c x
    m >>= k = λc → m (λa → k a c)
```

Identity
(Modelliert: reine Funktionen)

```
type Identity a = a
instance Monad Identity where
    return = id
    m >>= k = k m
```

IO
(Modelliert: Berechnungen, die mit der „echten Welt“ kommunizieren, etwa Dateizugriffe. Implementierung ist Compiler-Geheimnis!)

```
type IO = ?
instance Monad IO where
    return = ?
    m >>= k = ?
```

Do-Notation
Haskell unterstützt eine spezielle Syntax für monadische Berechnungen. So steht etwa

```
main = do x ← getLine
          putStr ("You said " ++ x)
          return 3
```

für

```
main = getLine >>= (λx →
       (putStr ("You said " ++ x) >>= (λ_ →
       return 3))).
```