Funktionale Programmierung

Funktoren und Monaden

Andreas Abel und Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

23. April 2012

Funktoren

- Typ-Ausdrücke mit Parametern bezeichnen wir als *Typkonstruktoren*.

  type *List a* = [*a*]

- Der Aufruf *map f l* wendet *f* auf jedes Element der Liste *l* an.

  $map :: (a \to b) \to [a] \to [b]$

- Die Existenz der *map*-Funktion macht den Typkonstruktor [] für Listen zu einem *Funktor*.
- Ein Funktor ist eine Abbildung von einer Kategorie in eine andere (oder diesselbe) Kategorie.
- Im Falle von [] ist die Kategorie jeweils die Menge der Typen.
- vgl. SML-Funktoren: bilden Strukturen auf Strukturen ab. Kategorie ist jeweils eine Signatur.

Funktoren in Haskell

- Typ(konstruktor)klasse *Functor* klassifiziert Funktoren.

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

instance Functor List where
  fmap = map

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

- Jede *collection* ist ein Funktor!
- Funktorialität kann in GHC automatisch hergeleitet werden.

```
{-# LANGUAGE DeriveFunctor #-}

data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Functor)
```

Funktorengesetze

- Für die Aktion *fmap* eines Funktors müssen 2 Gesetze gelten:
  1. Identität:

     $$fmap\ id = id$$

  2. Komposition:

     $$fmap\ f \circ fmap\ g = fmap\ (f \circ g)$$

- Diese Gesetze werden von Haskell *nicht geprüft*!
- Fehlerhafte Definition von *fmap* für *Tree*:

```
instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node a l r) = Node (f a) Leaf (fmap f r)
```

- Welches Gesetz wird gebrochen?

Beweis durch strukturelle Induktion

- Das Kompositions-Gesetz gilt für *map* für beliebige f, g, xs:

$$map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$$

- Beweis durch strukturelle Induktion über Listen.
  1. Fall $xs = []$. Da $map\ h\ [] = []$ für beliebige Funktionen h sind beide Seiten gleich.
  2. Fall $x : xs$.

$$\begin{aligned} map\ f\ (map\ g\ (x : xs)) &= map\ f\ (g\ x : map\ g\ xs) \\ &= f\ (g\ x) : map\ f\ (map\ g\ xs) \\ map\ (f \circ g)\ (x : xs) &= f\ (g\ x) : map\ (f \circ g)\ xs \end{aligned}$$

Nach Induktionsvoraussetzung ist
$map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$. Also sind beide Seiten gleich.

Zusammenfassung Funktoren

- *Funktor* ist ein Programmierschema für punktweise Anwendung einer Funktion an jeder Stelle einer Struktur.
- Die *funktorielle Aktion fmap* ist eine Generalisierung der *map*-Funktion von Listen auf andere Datenstrukturen.
- Identitätsgesetz: *fmap* verändert nie die Form, nur den Inhalt einer Datenstruktur.
- Kompositionsgesetz: Änderungen sind lokal und voneinander unabhängig.
- Übung: Zeigen Sie, dass in einer effektbehafteten Sprache wie SML das Kompositionsgesetz für *List.map* nicht im Allgemeinen gilt!

Monaden

- Eine *Monade* ist ein Programmier-Schema für sequentielle Berechnungen.
- In Haskell werden Berechnungen mit Effekten (wie Zustand, Ein-/Ausgabe, Ausnahmen) mittels Monaden simuliert.
- Monaden sind *keine* Spracherweiterung, man könnte alles rein funktional zu Fuß programmieren.
- Jedoch gibt es syntaktische Unterstützung für Monaden, jedoch auch die selbst-programmierten Monaden.
- Sie helfen lediglich bei der Strukturierung von Code (Modularität, Lesbarkeit).
- Die Schnittstelle zur *Außenwelt* läuft über die *IO*-Monade.

Parser als effektvolle Berechnung

1. Zustand: Die Eingabe wird um den geparsten Teil verkürzt.

   $$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to (a, \mathit{String})$$

   In der rein funktionalen Welt wird der Zustand *durchgeschleift*, ist also Eingabe und Ausgabe des Parsers.

2. Nicht-Determinismus: Mehrere Ergebnisse sind möglich, realisiert als Liste.

   $$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

   Wir verwenden nur zwei mögliche Ergebnisse: Die leere Liste $[]$ ist *Ausnahme*, die einelementige $[(a, \mathit{inp}')]$ ist reguläres Ergebnis.

Parser als sequentielle Berechnung

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

Zwei Parser werden mit *bindP* in Sequenz geschaltet.

$$\begin{aligned}
&\mathit{bindP} && :: \mathit{Parser}\ a \to (a \to \mathit{Parser}\ b) \to \mathit{Parser}\ b \\
&\mathit{bindP}\ p\ k = \lambda \mathit{inp} \to \textbf{case}\ (p\ \mathit{inp})\ \textbf{of} \\
&\qquad\qquad [\,] \to [\,] \\
&\qquad\qquad [(a, \mathit{inp}')] \to (k\ a)\ \mathit{inp}'
\end{aligned}$$

returnP ist völlig effekt*neutral*, es kann keine Aufnahme werfen und gibt die Eingabe ungesehen weiter.

$$\begin{aligned}
&\mathit{returnP} && :: a \to \mathit{Parser}\ a \\
&\mathit{returnP}\ a = \lambda \mathit{inp} \to [(a, \mathit{inp})]
\end{aligned}$$

Gesetze der Sequenz

1. Links-Eins:

$$(returnP\ a)\ `bindP`\ (\lambda a' \rightarrow k\ a') = k\ a$$

2. Rechts-Eins:

$$p\ `bindP`\ (\lambda a \rightarrow returnP\ a) = p$$

3. Assoziativität:

$$(p\ `bindP`\ \lambda a \rightarrow q\ a)\ `bindP`\ \lambda b \rightarrow k\ b =$$
$$p\ `bindP`\ \lambda a \rightarrow (q\ a\ `bindP`\ \lambda b \rightarrow k\ b)$$

Monaden-Definition

Eine *Monade* ist ein Tripel $(M, return, (\gg\!=))$ (sprich "*bind*" für $\gg\!=$) mit folgender Typisierung:

$$\textbf{type } M\ a$$
$$return :: a \to M\ a$$
$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$

Folgende Gesetze müssen gelten:

$$\begin{array}{lll} return\ a \gg\!= k & = k\ a & \text{-- Links-Eins} \\ m \gg\!= return & = m & \text{-- Rechts-Eins} \\ (m \gg\!= \lambda a \to k\ a) \gg\!= k' = & & \text{-- Assoziativität} \\ m \gg\!= \lambda a \to (k\ a \gg\!= k') & & \end{array}$$

Monade ist abgeleitet von *Monoid*. In der Kategorientheorie bezeichnet man $(M, return, (\gg\!=))$ als *Kleisli-Tripel*.

Fehler-Monade

Eine Berechnung liefert entweder ein Ergebnis a oder wirft die Ausnahmne s.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Except}\ a &&= \mathit{Fail}\ \mathit{String}\\
&&&\mid \mathit{Ok}\ \ a\\
&\mathit{returnE} &&:: a \to \mathit{Except}\ a\\
&\mathit{returnE}\ a &&= \mathit{Ok}\ a\\
&\mathit{bindE} &&:: \mathit{Except}\ a \to (a \to \mathit{Except}\ b) \to \mathit{Except}\ b\\
&\mathit{bindE}\ (\mathit{Fail}\ s)\ k &&= \mathit{Fail}\ s\\
&\mathit{bindE}\ (\mathit{Ok}\ a)\ \ k &&= k\ a
\end{aligned}
$$

Variante mit parametrisiertem Fehlertyp:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Except}\ s\ a = \mathit{Fail}\ s\\
&\qquad\qquad\qquad\mid \mathit{Ok}\ \ a
\end{aligned}
$$

In der Praxis verwendet man hierfür *Either*.

Beispiel: Division

```
divideE       :: Except Int → Except Int → Except Int
divideE m n = m `bindE` (λx →
                n `bindE` (λy →
                  if y ≡ 0 then Fail "division by zero"
                           else returnE (x `div` y)))
```

In Haskell kann man das viel leserlicher schreiben.

```
divideE       :: Except Int → Except Int → Except Int
divideE m n = do x ← m
                 y ← n
                 if y ≡ 0 then fail "division by zero"
                          else return (x `div` y)
```

Sieht schon fast aus wie ein imperatives Programm!

do-Notation

Um die **do**-Notation benutzen zu können, muss man Haskell mitteilen, dass *Except* eine Monade ist:

instance *Monad Except* **where**
$$\begin{aligned} return &= returnE \\ (\gg\!=) &= bindE \\ fail &= Fail \end{aligned}$$

1. Setzt das überladene *return*.
2. Setzt das überladene $\gg\!=$ und installiert die **do**-Notation.
3. (optional) Setzt die Null der Monade auf *Fail*. Standardmäßig ist sie *error*. Für die Null gilt:

$$fail\ s \gg\!= k = fail\ s$$

Weiteres Aufhübschen

Haskell definiert:

```
($)    :: (a -> b) -> a -> b
f $ a = f a
```

Und im Modul *Control.Monad*:

```
when          :: Monad m => Bool -> m () -> m ()
when True  m = m
when False m = return ()
```

Damit reduzieren wir Schachtelung und Klammern:

```
divideE       :: Except Int -> Except Int -> Except Int
divideE m n = do x <- m
                 y <- n
                 when (y == 0) $ fail "division by zero"
                 return $ x `div` y
```

Das ist doch kein Zustand!

Wir wollen eine Liste umdrehen und dabei zählen wieviele Nullen sie enthält.

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Count}\ a = \mathit{Int} \to (a, \mathit{Int}) \\
\mathit{inc} & :: & \mathit{Count}\ () \\
\mathit{inc} & = & \lambda n \to ((), n+1) \\
\mathit{returnC} & :: & a \to \mathit{Count}\ a \\
\mathit{returnC}\ a & = & \lambda n \to (a, n) \\
\mathit{bindC} & :: & \mathit{Count}\ a \to (a \to \mathit{Count}\ b) \to \mathit{Count}\ b \\
\mathit{bindC}\ m\ k & = & \lambda n \to \textbf{let}\ (a, n') = m\ n \\
 & & \qquad\qquad \textbf{in}\ \ k\ a\ n'
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Count}\ \textbf{where} \\
\quad \mathit{return} = \mathit{returnC} \\
\quad (\gg\!=) = \mathit{bindC}
\end{array}
$$

Haskell meckert...

Zustand

Haskell nötigt uns zu einer **newtype**-Deklaration. Das ist ein **data** mit genau einem Konstruktor, der wiederum genau ein Argument nimmt.

$$
\begin{array}{ll}
\textbf{newtype}\ \mathit{Count}\ a = \mathit{C}\ (\mathit{Int} \to (a, \mathit{Int})) & \\
\mathit{runC} & :: \mathit{Count}\ a \to \mathit{Int} \to (a, \mathit{Int}) \\
\mathit{runC}\ (\mathit{C}\ f)\ n & = f\ n \\
\mathit{inc} & :: \mathit{Count}\ () \\
\mathit{inc} & = \mathit{C}\ \$\ \lambda n \to ((), n+1) \\
\mathit{returnC} & :: a \to \mathit{Count}\ a \\
\mathit{returnC}\ a & = \mathit{C}\ \$\ \lambda n \to (a, n) \\
\mathit{bindC} & :: \mathit{Count}\ a \to (a \to \mathit{Count}\ b) \to \mathit{Count}\ b \\
\mathit{bindC}\ m\ k & = \mathit{C}\ \$\ \lambda n \to \textbf{let}\ (a, n') = \mathit{runC}\ m\ n \\
 & \qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{runC}\ (k\ a)\ n'
\end{array}
$$

instance *Monad Count* **where**

Reverse mit Nullenzähler

Die endrekursive Definition von *reverse*, monadisch, mit Zähler:

$$
\begin{aligned}
&\mathit{revApp} && :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Count}\ [\mathit{Int}] \\
&\mathit{revApp}\ [\,] && \mathit{acc} = \mathit{return}\ \mathit{acc} \\
&\mathit{revApp}\ (x : \mathit{xs})\ \mathit{acc} && = \mathbf{do}\ \mathit{when}\ (x \equiv 0)\ \$\ \mathit{inc} \\
& && \qquad \mathit{revApp}\ \mathit{xs}\ (x : \mathit{acc}) \\
&\mathit{revCountZeros} && :: [\mathit{Int}] \to ([\mathit{Int}], \mathit{Int}) \\
&\mathit{revCountZeros}\ \mathit{l} && = \mathit{runC}\ (\mathit{revApp}\ \mathit{l}\ [\,])\ 0
\end{aligned}
$$

Test:

$$\mathit{revCountZeros}\ [0,0,1,0,1,2,0,1,2,3]$$
$$([3,2,1,0,2,1,0,1,0,0],4)$$

Parametrisierte Zustands-Monade

Verallgemeinerung von einem *Int*-Zähler auf beliebigen Zustand s.

$$\begin{aligned}
&\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\}\\
&\mathit{returnSt} && :: a \to \mathit{State}\ s\ a\\
&\mathit{returnSt}\ a && = \mathit{State}\ \$\ \lambda s \to (a, s)\\
&\mathit{bindSt} && :: \mathit{State}\ s\ a \to (a \to \mathit{State}\ s\ b) \to \mathit{State}\ s\ b\\
&\mathit{bindSt}\ m\ k && = \mathit{State}\ \$\ \lambda s \to \textbf{let}\ (a, s') = \mathit{runState}\ m\ s\\
&&&\qquad\qquad\ \ \textbf{in}\ \ \mathit{runState}\ (k\ a)\ s'
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\
&\quad \mathit{return} = \mathit{returnSt}\\
&\quad (\gg\!=)\ = \mathit{bindSt}
\end{aligned}$$

Eine **data**-Deklaration kann auch gleich Destruktoren mitdefinieren. Im Falle von **newtype** ist es genau ein Destruktor.

$$\begin{aligned}
&\mathit{runState} && :: \mathit{State}\ s\ a \to s \to (a, s)\\
&\mathit{runState}\ (\mathit{State}\ f)\ s = f\ s
\end{aligned}$$

IO-Monade

Die Kommunikation zur Außenwelt erfolg in Haskell über die IO-Monade (*input/output*). Zum Beispiel:

instance *Monad IO*

```
putChar    :: Char → IO ()    -- put char. on stdio
putStr     :: String → IO ()  -- put string on stdio
putStrLn   :: String → IO ()  -- put string + newline on stdio
getChar    :: IO Char         -- read char. from stdio
getLine    :: IO String       -- read line from stdio
getContents :: IO String      -- read whole input from stdio
```

Weiteres: Datei-Ein-Ausgabe.

IO: Kommunikation mit der Welt

- Ein IO-Effekt verändert den Zustand der Welt, die das Programm umgibt.
- Beispiel: Lesen in einer Datei (setzt den Lesezeiger weiter).
- Man kann sich *IO* als *State World* vorstellen:

$$\textbf{newtype}\ \mathit{IO}\ a = \mathit{IO}\ \{\mathit{runIO} :: \mathit{World} \to (a, \mathit{World})\}$$

- Allerdings gibt es kein *runIO*: Den Zustand der Welt kann man nicht lesen und setzen!
- *IO* ist notwendigerweise eine Primitive.

My first I/O-Program in Haskell

- Ein ausführbares (standalone) Programm benötigt eine Funktion (entry point)

$$main :: IO\ ()$$

- Die "Kopier-Katze" (copy cat), naiv implementiert.

$$\begin{aligned} &main :: IO\ () \\ &main = \mathbf{do}\ c \leftarrow getChar \\ &\qquad\qquad\ putChar\ c \\ &\qquad\qquad\ main \end{aligned}$$

Zusammenfassung Monaden

- Monaden: Konzept aus der Kategorientheorie (MacLane, 1971) und der Programmiersprachentheorie (Moggi, 1991)
- Modelliert Ein-/Ausgabe und andere effektvolle Berechnungen
- Gibt es in: Haskell, F#, Scala...
- Ausblick:
  - Listenmonade für Nichtdeterminismus
  - Lesemonade zum Verstecken von Funktionsparametern
  - Schreibemonade für Ergebnisakkumulation
  - Monadenstapel und -transformatoren
  - Imperative Programmierung mit Monaden
- Übungen:
  - Zeigen Sie: *Maybe* ist eine Monade!
    Beweisen Sie auch die Monadengesetze!
  - Zeigen Sie: Jede Monade ist ein Funktor!

Literatur zu Monaden

- Phil Wadler, Monads for functional programming (1992)
- Wikipedia-Artikel zu Monads(functional programming)
- `http://en.wikibooks.org/wiki/Haskell/Understanding_monads`
- Haskell Bibliotheken:

```
import Control.Monad
import Control.Monad.Error   -- package mtl
import Control.Monad.State   -- package mtl

import System.IO
import System.Environment
import System.Exit
```