3.– 6. September 2012
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Monaden

Die nächste Herausforderung der funktionalen Welt

Nicole Rauch
msgGillardon AG

Überblick

- Grundsätzliche Funktionsweise von Monaden
- Beispielhafte Vorstellung mehrerer Monaden
- Monaden in Java?!

Warum funktionale Programmierung?

Warum funktionale Programmierung?

- Höhere Abstraktionsmöglichkeiten
  - z. B. Higher Order Functions, Closures
  - ermöglichen knappen, klaren Code

Warum funktionale Programmierung?

- Höhere Abstraktionsmöglichkeiten
  - z. B. Higher Order Functions, Closures
  - ermöglichen knappen, klaren Code
- Besseres Laufzeitverhalten
  - durch Lazy Evaluation, d. h. Auswertung von Funktionsargumenten erst beim tatsächlichen Zugriff

Warum funktionale Programmierung?

- Höhere Abstraktionsmöglichkeiten
  - z. B. Higher Order Functions, Closures
  - ermöglichen knappen, klaren Code
- Besseres Laufzeitverhalten
  - durch Lazy Evaluation, d. h. Auswertung von Funktionsargumenten erst beim tatsächlichen Zugriff
- Parallelisierung
  - angesichts immer größerer Prozessorzahlen zunehmend wichtiger

Warum funktionale Programmierung?

- Höhere Abstraktionsmöglichkeiten
  - z. B. Higher Order Functions, Closures
  - ermöglichen knappen, klaren Code
- Besseres Laufzeitverhalten
  - durch Lazy Evaluation, d. h. Auswertung von Funktionsargumenten erst beim tatsächlichen Zugriff
- Parallelisierung
  - angesichts immer größerer Prozessorzahlen zunehmend wichtiger
- Continuations
  - Beschreibung des Ablaufzustands eines Programms
  - z. B. zur Live-Migration laufender Prozesse

Warum funktionale Programmierung?

- Höhere Abstraktionsmöglichkeiten
  - z. B. Higher Order Functions, Closures
  - ermöglichen knappen, klaren Code
- Besseres Laufzeitverhalten
  - durch Lazy Evaluation, d. h. Auswertung von Funktionsargumenten erst beim tatsächlichen Zugriff
- Parallelisierung
  - angesichts immer größerer Prozessorzahlen zunehmend wichtiger
- Continuations
  - Beschreibung des Ablaufzustands eines Programms
  - z. B. zur Live-Migration laufender Prozesse
- Einfache Einbettung domänenspezifischer Sprachen
  - inklusive Typsicherheit

Und was sind jetzt Monaden?

Monaden

- sind ein Abstraktionskonzept
- vereinheitlichen unterschiedliche Aspekte
- kapseln immer wiederkehrende Aspekte
- helfen dadurch beim Fokussieren
- schlagen (ganz nebenbei) eine Brücke zwischen der funktionalen und der nichtfunktionalen Welt

Drei einfache Funktionen

```
Integer minus5(Integer x) {
    if (x == 7) {
        return null;
    }
    return x - 5;
}

Integer mal3(Integer x) {
    if (x % 2 == 0) {
        return null;
    }
    return x * 3;
}

Integer plus7(Integer x) {
    return x + 7;
}
```

Die Funktionen hintereinandergeschaltet

```
minus5 (mal3 (plus7 ( 12 )))
```

liefert 52

Die Funktionen hintereinandergeschaltet

```
minus5 (mal3 (plus7 ( 12 )))
```

liefert 52

```
minus5 (mal3 (plus7 ( 1 )))
```

liefert eine Exception...

Die Funktionen hintereinandergeschaltet

```
minus5 (mal3 (plus7 ( 12 )))
```

liefert 52

```
minus5 (mal3 (plus7 ( 1 )))
```

liefert eine Exception...

(wir erinnern uns: `mal3` mag nur ungerade Zahlen)

Also entweder besser aufpassen...

```
Integer aufpassen(Integer x1) {
    if (x1 != null) {
        Integer x2 = plus7(x1);
        if (x2 != null) {
            Integer x3 = mal3(x2);
            if (x3 != null) {
                return minus5(x3);
            }
        }
    }
    return null;
}
```

...oder robustere Funktionen bauen?

```
Integer minus5(Integer x) {
    if (x == null || x == 7) {
        return null;
    }
    return x - 5;
}

Integer mal3(Integer x) {
    if (x == null || x % 2 == 0) {
        return null;
    }
    return x * 3;
}

Integer plus7(Integer x) {
    if (x == null) { return null; }
    return x + 7;
}
```

Der Effekt

- Die null-Prüfungen müssen überall eingefügt werden
- Vergessene Prüfungen führen zu Laufzeitfehlern
- Der Prüf-Code lenkt von der eigentlichen Fachlichkeit ab

Die drei Funktionen in Haskell

Das Haskell-Äquivalent zu nullable Types:

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

Die drei Funktionen in Haskell

Das Haskell-Äquivalent zu nullable Types:

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

Unsere Funktionen in Haskell:

```
minus5 Nothing = Nothing
minus5 (Just x)
    | x == 7    = Nothing
    | otherwise = Just (x - 5)

mal3 Nothing = Nothing
mal3 (Just x)
    | even x    = Nothing
    | otherwise = Just (3 * x)

plus7 Nothing = Nothing
plus7 (Just x) = Just (x + 7)
```

Separation of Concerns

Unsere Funktionen sollen sich nur um die Fachlichkeit kümmern:

```
minus5 x
    | x == 7    = Nothing
    | otherwise = Just (x - 5)

mal3 x
    | even x    = Nothing
    | otherwise = Just (3 * x)

plus7 x = Just (x + 7)
```

Auslagern der Null-Behandlung

Das Verbinden der Funktionen lagern wir aus:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x
```

Auslagern der Null-Behandlung

Das Verbinden der Funktionen lagern wir aus:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x
```

Der initiale Einstieg in den Maybe-Typ:

```
return :: a -> Maybe a

return x  =  Just x
```

Auslagern der Null-Behandlung

Das Verbinden der Funktionen lagern wir aus:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x
```

Der initiale Einstieg in den Maybe-Typ:

```
return :: a -> Maybe a

return x  =  Just x
```

Unsere Methodenaufrufe:

```
return 12 >>= plus7 >>= mal3 >>= minus5
```

Wann kommen denn endlich die Monaden?

Wann kommen denn endlich die Monaden?

Einen Datentyp, zum Beispiel

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

Wann kommen denn endlich die Monaden?

Einen Datentyp, zum Beispiel

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

auf dem zwei Funktionen

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

return :: a -> Maybe a
```

definiert sind,

Wann kommen denn endlich die Monaden?

Einen Datentyp, zum Beispiel

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

auf dem zwei Funktionen

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

return :: a -> Maybe a
```

definiert sind,

nennt man eine Monade! *

Vorteile und Nachteile

Vorteile von Monaden:

- Boilerplate Code ist wegabstrahiert
- Höheres konzeptionelles Niveau des Codes
- Es gibt Funktionsbibliotheken, die auf Monaden aufbauen
- Verschiedene Monaden lassen sich miteinander kombinieren (Monaden-Transformer)

Vorteile und Nachteile

Vorteile von Monaden:

- Boilerplate Code ist wegabstrahiert
- Höheres konzeptionelles Niveau des Codes
- Es gibt Funktionsbibliotheken, die auf Monaden aufbauen
- Verschiedene Monaden lassen sich miteinander kombinieren (Monaden-Transformer)

Nachteile von Monaden:

- Größere Einstiegshürde und Lernkurve
- Benötigt implizites Wissen über die Funktionsweise

Die einfachste Monade

Der Datentyp:

```
a
```

Die Funktionen:

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

x >>= f = f x
```

```
return :: a -> a

return x = x
```

Exception-Handling

```
public int throwAt7(int number){
    if( number == 7 ){
        throw new IllegalArgumentException("Ich hasse 7!");
    }

    return number + 1;
}
```

In rein funktionalen Sprachen ist dies so nicht möglich!

Eine Monade als Alternative zu Exceptions

Der Datentyp:

```
data Exceptional e a =
     Success a
   | Exception e
   deriving (Show)
```

enthält entweder eine Exception oder die Rückgabe.

Eine Monade als Alternative zu Exceptions

Der Datentyp:

```
data Exceptional e a =
     Success a
   | Exception e
   deriving (Show)
```

enthält entweder eine Exception oder die Rückgabe.
Die Funktionen:

```
(>>=) :: Exceptional e s -> (s -> Exceptional e s) -> Exceptional e s

Exception exc >>= _    =  Exception exc
Success x     >>= f    =  f x
```

```
return :: s -> Exceptional e s

return = Success
```

throw und catch

```
throw :: e -> Exceptional e s

throw  =  Exception
```

```
catch :: Exceptional e s -> (e -> Exceptional e s) -> Exceptional e s

catch (Exception  exc) f  =  f exc
catch (Success x)      _  =  Success x
```

Anwendungsbeispiel der Exception-Monade

```
public int throwAt7(int x){
    if( x == 7 ){
        throw new IllegalArgumentException("Ich hasse 7!");
    }

    return x + 1;
}
```

Anwendungsbeispiel der Exception-Monade

```
public int throwAt7(int x){
    if( x == 7 ){
        throw new IllegalArgumentException("Ich hasse 7!");
    }

    return x + 1;
}
```

```
data Exceptions = IllegalArgumentException String

throwAt7 x
   | x == 7     =  throw (IllegalArgumentException "Ich hasse 7!")
   | otherwise  =  Success (x + 1)
```

Funktionsverknüpfung in der Exception-Monade

```
throwAt7 x
   | x == 7     =  throw (IllegalArgumentException "Ich hasse 7!")
   | otherwise  =  Success (x + 1)
```

```
mal3 :: Int -> Int
mal3 x  =  3 * x
```

Funktionsverknüpfung in der Exception-Monade

```
throwAt7 x
   | x == 7     =  throw (IllegalArgumentException "Ich hasse 7!")
   | otherwise  =  Success (x + 1)
```

```
mal3 :: Int -> Int
mal3 x  =  3 * x
```

Herstellen der passenden Signatur für mal3:

```
return . mal3
```

hat die Signatur

```
Int -> Exceptional e Int
```

Funktionsverknüpfung in der Exception-Monade

```
print $ (throwSomething 12) >>= (return . mal3)
```

liefert `Success 39`

Funktionsverknüpfung in der Exception-Monade

```
print $ (throwSomething 12) >>= (return . mal3)
```

liefert `Success 39`

Funktionsverknüpfung in der Exception-Monade

```
print $ (throwSomething 12) >>= (return . mal3)
```

liefert `Success 39`

```
print $ (throwSomething 7) >>= (return . mal3);
```

liefert `Exception (IllegalArgumentException "Ich hasse 7!")`

Vorteile

- Sprachunterstützung für Exceptions ist nicht erforderlich
- Keine versehentlichen Programmabbrüche durch Exceptions
- Volle Kontrolle über den Programmfluss, trotzdem kein Boilerplate-Code

System Output

```
public int numberAndText(){

    System.out.println("Hallo Herbstcampus!");

    return 7;
}
```

Eine Monade als Alternative zu System Output

Der Datentyp:

```
data Sysout a = ResOut a String
        deriving (Show)
```

soll sämtliche Systemausgaben aufsammeln.

Eine Monade als Alternative zu System Output

Der Datentyp:

```
data Sysout a = ResOut a String
        deriving (Show)
```

soll sämtliche Systemausgaben aufsammeln.

Die Funktionen:

```
(>>=) :: Sysout a -> (a -> Sysout b) -> Sysout b

(ResOut res1 out1) >>= f = let (ResOut res2 out2) = f res1
                           in  (ResOut res2 (out1 ++ out2))
```

```
return :: a -> Sysout a

return x = ResOut x ""
```

Anwendungsbeispiel

```
public int numberAndText(){

    System.out.println("Hallo Herbstcampus!");

    return 7;
}
```

```
numberAndText = ResOut 7 "Hallo Herbstcampus!"
```

Anwendungsbeispiel

```
public int numberAndText(){

    System.out.println("Hallo Herbstcampus!");

    return 7;
}
```

```
numberAndText = ResOut 7 "Hallo Herbstcampus!"
```

```
public int calcAndText(int x){

    System.out.println("Ich kann rechnen!");

    return 4 + x;
}
```

```
calcAndText x = ResOut (4 + x) "Ich kann rechnen!"
```

Funktionsverknüpfung

```
print (numberAndText >>= calcAndText)
```

liefert `ResOut 11 "Hallo Herbstcampus! Ich kann rechnen!"`

Vorteile

- System Output bleibt stets in der richtigen Reihenfolge, auch bei Parallelisierung
- Die tatsächliche Textausgabe ist nicht mehr über den ganzen Code verteilt, sondern zentral gebündelt

Grundsätzliches

- Es gibt keine Seiteneffekte $\Rightarrow$ Funktionen problemlos testbar
- Man kann sofort erkennen, ob eine Funktion eine bestimmte Monade einsetzt (z. B. einen bestimmten Seiteneffekt hat)
- Manche Monaden garantieren, dass alle Ausdrücke nacheinander ausgeführt werden, keine Vertauschung der Ausführungsreihenfolge durch lazy evaluation

Monaden in Java

Die Maybe-Monade:

```
Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x

return x  =  Just x
```

Monaden in Java

Die Maybe-Monade:

```
Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x

return x  =  Just x
```

wird für den Maybe-Typ `Integer` zu

```
interface Func {
    Integer eval(int arg);
}
```

```
Integer bind(Integer arg, Func f) {
    if( arg == null ) return null;
    return f.eval(arg.intValue());
}

Integer _return( int arg ){   return Integer.valueOf(arg);   }
```

Monadische Funktionsverknüpfung in Java

Aus

```
return 12 >>= plus7 >>= mal3 >>= minus5
```

wird

Monadische Funktionsverknüpfung in Java

```
bind(bind(bind(_return(12), new Func() {

    @Override
    public Integer eval(int arg) {
        return plus7(arg);
    }
}), new Func() {

    @Override
    public Integer eval(int arg) {
        return mal3(arg);
    }
}), new Func() {

    @Override
    public Integer eval(int arg) {
        return minus5(arg);
    }
});
```

Vielen Dank!

Code & Folien auf GitHub:

`https://github.com/NicoleRauch/Monaden`

Nicole Rauch

E-Mail `nicole.rauch@msg-gillardon.de`

Twitter `@NicoleRauch`