5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Monaden & Co.

Funktionale Konzepte in Scala

Lars Hupel

Monaden & Co.

Funktionale Konzepte in Scala
Herbstcampus 2011 – S24

Lars Hupel
hupel@in.tum.de

7. September 2011

Über den Autor

- Informatik-Student, TU München
- Java-Programmierer seit 5 Jahren
- Scala-Programmierer seit 2 Jahren
- Interesse an formalen Methoden, Typsystemen, ...

Werbung

Scala Enthusiasten Metropolregion Nürnberg

Scalas Typsystem

- Grundlage bleibt objektorientiert
- erweiterte Typinferenz
- Singletons
- Existential types
- Definition-site variance
- Implizite Konversionen und Parameter
- ...

separater Vortrag dazu („Schwarze Magie“)

Exkurs: Haskell

- Begriff „Typklasse“ hat nichts mit „Klasse“ im Java-Sinne zu tun
- bedeutet eher „Familie“ von Typen, die gemeinsame Operationen anbieten

Exkurs: Haskell

Beispiel

Zahloperationen

```
class Numeric a where
    add :: a -> a -> a
    neg :: a -> a
    zero :: a
```

Exkurs: Haskell

Implementation

Datentypen stellen Implementationen von Klassen bereit

```
instance Numeric (Float, Float) where
    add (x1, y1) (x2, y2) = (x1+x2, y1+y2)
    neg (x, y) = (-x, -y)
    zero = (0.0, 0.0)
```

Exkurs: Haskell

Implementation

Implementationen können auch von anderen abhängig sein

```
instance (Numeric a, Numeric b) =>
         Numeric (a, b) where
    add (x1, y1) (x2, y2) = (add x1 x2,add y1 y2)
    neg (x, y) = (neg x, neg y)
    zero = (zero, zero)
```

Exkurs: Haskell

Verwendung

generische Funktionen können Typen einschränken

```
sum :: Numeric a => [a] -> a
sum [] = zero
sum (x : xs) = add x (sum xs)
```

$\rightsquigarrow$ der Funktion wird eine Menge von Operationen „implizit“ bereitgestellt

```
> sum [(1.0, 2.0), (3.0, 4.0)]
(4.0,6.0)
```

Typklassen in Scala

Modellierung der Klasse als Trait

```
trait Numeric[A] {
    def add(a1: A, a2: A): A
    def neg(a: A): A
    val zero: A
 }
```

Typklassen in Scala

Implementation

... als Wert

```
val floatNumeric = new Numeric[Float] {
    def add(a1: Float, a2: Float) = a1 + a2
    def neg(a: Float) = -a
    val zero: Float = 0.0f
}
```

Typklassen in Scala

Implementation

... oder als Funktion

```
def pairNumeric[A, B](na: Numeric[A], nb: Numeric[B]) =
    new Numeric[(A, B)] {
        def add(a1: (A, B), a2: (A, B)) = // ...
    }
```

Typklassen in Scala

Verwendung

Funktionen erhalten zusätzlichen Parameter

```
def sum[A](list: List[A], num: Numeric[A]): A =
    if (list.isEmpty)
        num.zero
    else
        num.add(list.head, sum(list.tail, num))
```

aber: noch nicht ganz am Ziel

Typklassen in Scala

Verwendung

Funktionen erhalten zusätzlichen Parameter

```
def sum[A](list: List[A], num: Numeric[A]): A =
    if (list.isEmpty)
        num.zero
    else
        num.add(list.head, sum(list.tail, num))
```

aber: noch nicht ganz am Ziel

Implicits

Schlüsselwort `implicit` mit zwei Mechanismen

- Konversion: automatisches Aufrufen einer Umwandlungsfunktion, falls ein anderer Typ erwartet wird
- Parameter: automatisches Einsetzen eines Parameters passenden Typs

sehr gut geeignet, um Code-Duplikation zu verringern

Implicits

Schlüsselwort `implicit` mit zwei Mechanismen

- Konversion: automatisches Aufrufen einer Umwandlungsfunktion, falls ein anderer Typ erwartet wird
- Parameter: automatisches Einsetzen eines Parameters passenden Typs

sehr gut geeignet, um Code-Duplikation zu verringern

Implizite Konversionen

... können z. B. bestehenden Typen Methoden hinzufügen.

```
class ArrowAssoc[A](x: A) {
    def ->[B](y: B): (A, B) = (x, y)
}

implicit def any2ArrowAssoc[A](x: A) =
    new ArrowAssoc(x)

Map(1 -> "1", 2 -> "2")
// statt
Map((1, "1"), (2, "2"))
```

Implizite Parameter

... können lästiges Umherreichen von Parametern verkürzen

```
class Config {
    // ...
}

def doThat(param: Int)(implicit conf: Config) =
    println()

def doThis(param: Int)(implicit conf: Config) =
    doThat(param * 2)

doThis(3)(new Config())
```

Typklassen in Scala

Revisited

```
def sum[A](list: List[A])(implicit num: Numeric[A]): A =
    if (list.isEmpty)
        num.zero
    else
        num.add(list.head, sum(list.tail))
```

```
> sum(List((1.0f, 2.0f), (3.0f, 4.0f)))
res0: (Float, Float) = (4.0,6.0)
```

Typklassen für Container

- Beispiel: man verwendet mehrere Bibliotheken, die allesamt eigene (generische) Container-Typen verwenden
- manuelles Konvertieren ist fehleranfällig
- $\rightsquigarrow$ Typklassen, um gemeinsame Operationen anzubieten

Typklassen für Container

```
trait ListLike[T] {
    def toSet(list: T): Set[?]
}
```

Problem: Generizität $\rightsquigarrow$ Was setzt man als Typparameter ein?

Typklassen für Container

```
trait ListLike[T] {
    def toSet(list: T): Set[?]
}
```

Problem: Generizität $\rightsquigarrow$ Was setzt man als Typparameter ein?

Werte und Typen

- Typen klassifizieren Werte
- Was klassifiziert Typen?

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- $*$ steht für beliebigen Typ
- -> steht für Abbildung

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- $*$ steht für beliebigen Typ
- -> steht für Abbildung

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- * steht für beliebigen Typ
- -> steht für Abbildung

Typen und Kinds

Beispiele

Typ	Deklaration	Kind
Liste	`class List[A]`	`* -> *`
Paar	`class Tuple2[A, B]`	`(* x *) -> *`
Applikation	`type Ap[T[_], S] = T[S]`	`((* -> *) x *) -> *`

Typklassen für Container

Revisited

Lösung: Kind explizit spezifizieren

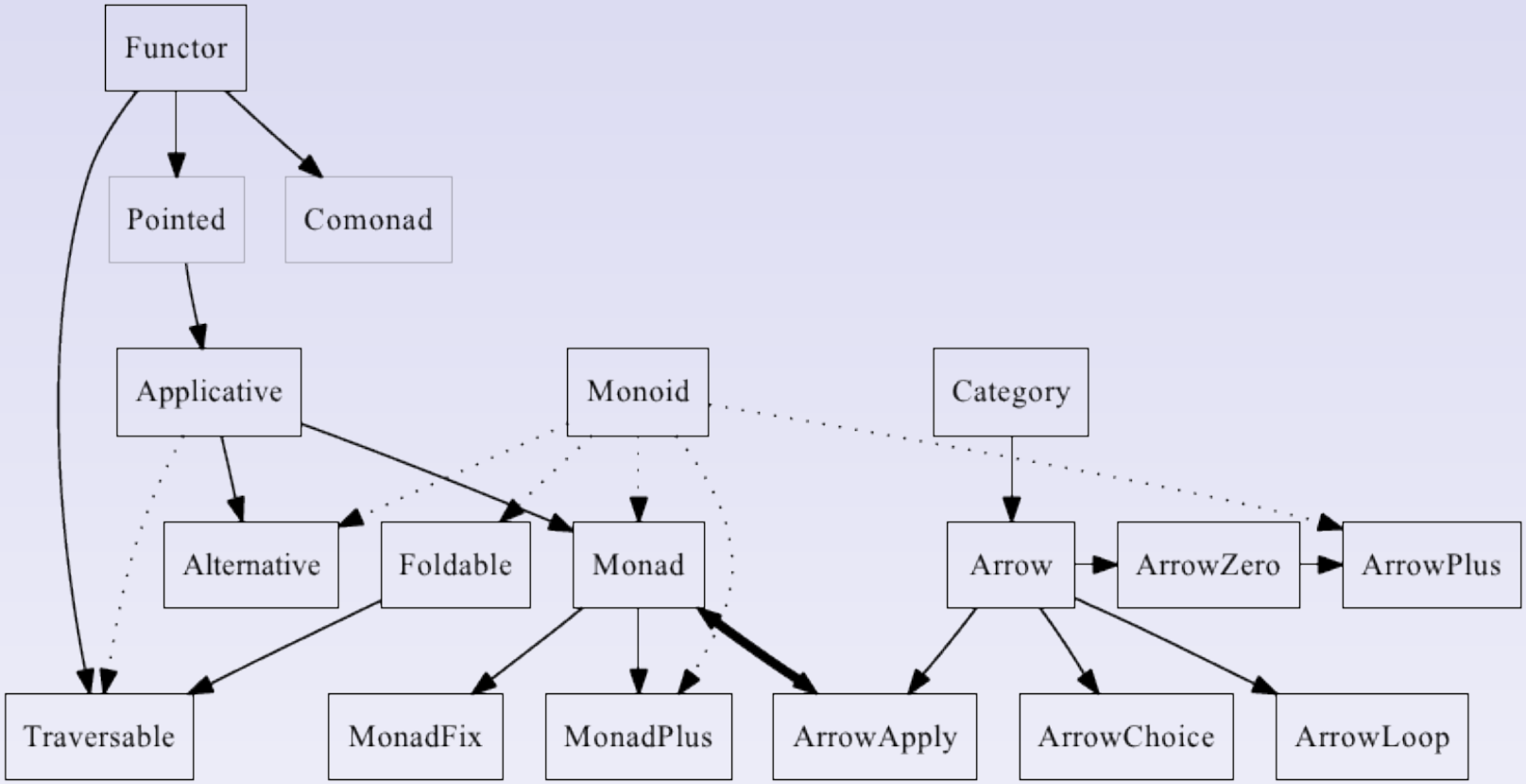
```
trait ListLike[T[_]] {
    def toSet[E](list: T[E]): Set[E]
}

implicit val list = new ListLike[scala.List] {
    def toSet[E](list: scala.List[E]) = list.toSet
}
```

Someone in the `#haskell` IRC channel used `(***)`, and when I asked `lambdabot` to tell me its type, it printed out scary gobbledygook that didn't even fit on one line! Then someone used `fmap fmap fmap` and my brain exploded.

– *aus "The Typeclassopedia" von Brent Yorgey*

```
> :type (***)
(Arrow a) =>
a b c -> a b' c' -> a (b, b') (c, c')
```

Haskells Typklassenhierarchie



aus “The Typeclassopedia”

Scalaz

The intention of Scalaz is to include general functions that are not currently available in the core Scala API.

– *Projektbeschreibung*

- besteht aus mehreren (unabhängigen) Teilen
- für diesen Vortrag von Interesse: Typklassen
- Nachbildung der Hierarchie aus Haskells Bibliothek

Functor

Intuition: „Container, der Elemente enthält“

Operationen

- `fmap`: Abbildung der Elemente vom Typ `A` auf Typ `B`

Bedingungen

- **Identität**
  `a == fmap(identity, a)`
- **Komposition**
  `fmap(f compose g, a) == fmap(f, fmap(g, a))`

Functor

Implementation

```
trait Functor[F[_]] {
    def fmap[A, B](f: A => B, r: F[A]): F[B]
}

implicit val optionFunctor =
    new Functor[Option] {
        def fmap[A, B](f: A => B, r: Option[A]) =
            r map f
    }
```

Functors

- Scalaz definiert Functors auch für JDK-Klassen(z. B. `Callable`)
- Vorteil: verschiedene Datenstrukturen lassen sich gleich behandeln

Polymorphism captures similar structure over different values, while type classes capture similar operations over different structures.

– *aus "The Haskell School of Expression" von Paul Hudak*

Pure

Intuition: „verpackt Wert in Container“

Operationen

- `pure`: Konvertieren eines Werts vom Typ `A` in einen Container des Typs `P[A]`

Pure

Implementation

```
trait Pure[P[_]] {
    def pure[A](a: => A): P[A]
}
```

Pure

Anwendungsbeispiel

aufwändige Berechnungen in ein `Future` verlegen

```
implicit val futurePure =
    new Pure[Future] {
        def pure[A](a: => A) = new FutureTask(
            new Callable[A] {
                def call = a
            }
        )
    }
```

Applicative

Intuition: „Berechnungskontext“

Operationen

- `Pure` + `Functor`
- `apply`: Anwenden einer Funktion, die selbst in den Container verpackt ist, auf einen Wert

die meisten Functors sind auch Applicatives

Applicative

Implementation

```
trait Applicative[Z[_]]
  extends Pure[Z] with Functor[Z] {
    def apply[A, B](f: Z[A => B], a: Z[A]): Z[B]
}
```

Applicative

Implementation

```
trait Applicative[Z[_]]
  extends Pure[Z] with Functor[Z] {
    def apply[A, B](f: Z[A => B], a: Z[A]): Z[B]
}
```

alternative Definition von `apply`

`Z[A => B] => (Z[A] => Z[B])`

$\rightsquigarrow$ „Berechnungskontext“

Applicative

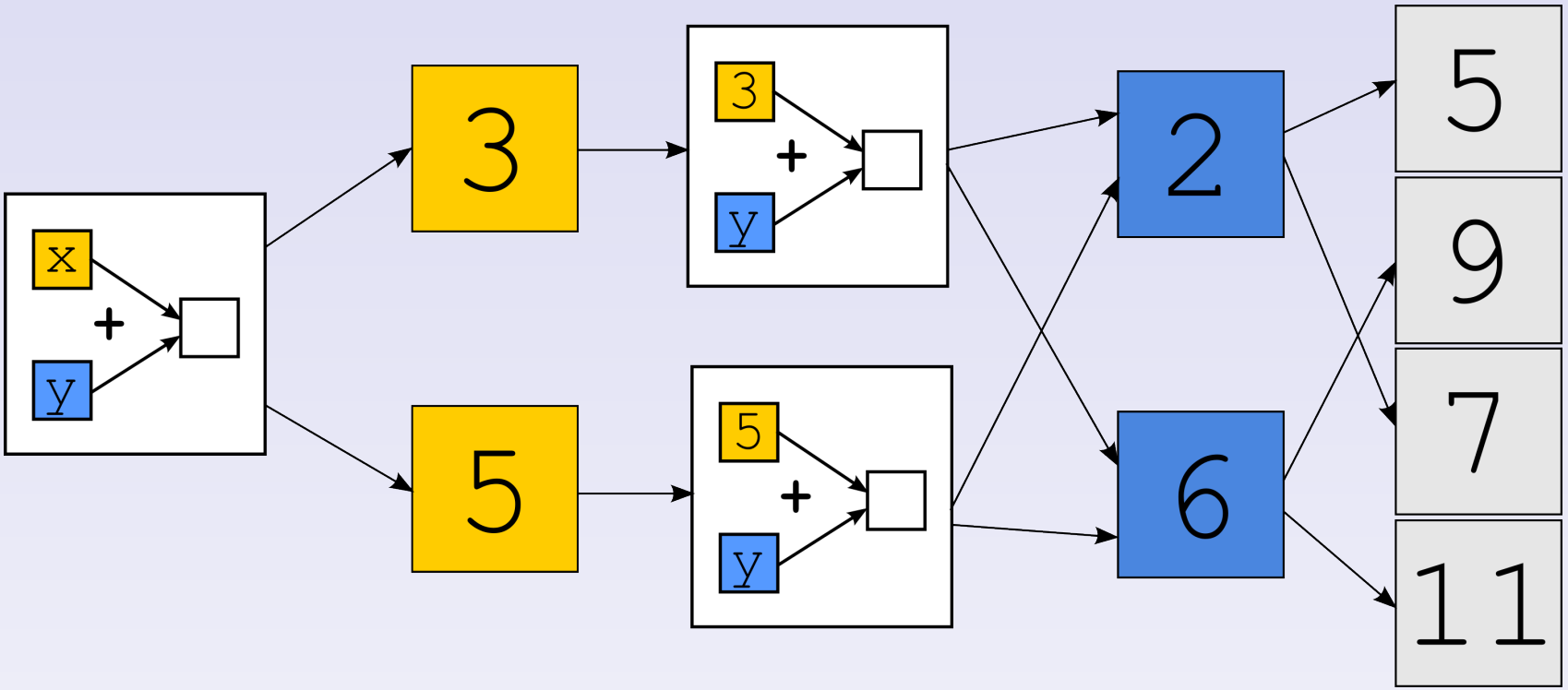
Beispiel: Listen

Idee: Listen repräsentieren Ergebnisse in einer nichtdeterministischen Berechnung

```
val nondet = new Applicative[List] {
    def pure[A](a: => A) = List(a)
    def apply[A, B](fs: List[A => B], as: List[A]) =
        for (f <- fs; a <- as)
            yield f(a)
}
```

Applicative

Beispiel: Listen




```
nondet.apply(nondet.fmap(f, List(3,5)), List(2,6))
```

Monad

Intuition: „kombiniert Berechnungen“

Operationen

- `Applicative`
- `flatMap`: verkettet Funktionen

Monad

Intuition: „kombiniert Berechnungen“

Operationen

- `Applicative`
- `flatMap`: verkettet Funktionen

Monad

A monad is just a monoid in the category of endofunctors, what's the problem?

– *James Iry (satirisch)*

Monad

Implementation

```
trait Monad[M[_]] extends Applicative[M] {
    def bind[A, B](a: M[A], f: A => M[B]): M[B]
}
```

Monad

Anwendungsbeispiel

Logging: Wert wird in mehreren Schritten errechnet, Teilergebnisse sollen protokolliert werden

Lösung: Modellierung als `(List[String], A)`

- `A`: aktuelles Teilergebnis
- `List[String]`: akkumuliertes Protokoll

Monad

Anwendungsbeispiel

Logging: Wert wird in mehreren Schritten errechnet, Teilergebnisse sollen protokolliert werden

Lösung: Modellierung als `(List[String], A)`

- `A`: aktuelles Teilergebnis
- `List[String]`: akkumuliertes Protokoll

Monad

The first law: a monad must not injure a computation or, through inaction, allow a computation to come to harm.

– *Paul Philipps (@extempore2)*

Monad

Beispiel: Logging

```
case class Computation[A](log: List[String], value: A)

val logger = new Monad[Computation] {
    def pure[A](a: => A) = Computation(Nil, a)
    def bind[A, B](a: Computation[A],
                   f: A => Computation[B]) = {
        val res = f(a.value)
        Computation(a.log ::: res.log, res.value)
    }
}
```

Monad

Beispiel: Logging

```
> logger.pure(5)
res0: (List[String], Int) = (List(),5)

> logger.bind(res0,
  (x: Int) => (
    List("adding to " + x),
    x + 3)
  )
res1: (List[String], String) =
  (List(adding to foo),8)
```

Probleme

Ursprüngliche Ziele

- wiederverwendbarer Code
- weniger Boilerplate

Probleme

Ursprüngliche Ziele

- wiederverwendbarer Code
- weniger Boilerplate

Probleme

Ursprüngliche Ziele

- wiederverwendbarer Code
- weniger Boilerplate

aber: `Monad` noch sehr unhandlich

In der Kürze...

vgl. Haskell:

```
log :: Int -> Writer [String] Int
log x = do a <- Writer (x + 3, ["adding to " ++ show x])
           return a

> runWriter (log 5)
(8,["adding to 5"])
```

Monads und Implicits

- Scala hat *for comprehensions*
- $\rightsquigarrow$ automatische Aufrufe von `map` und `flatMap`

Erkenntnis:

- `map` $\sim$ `fmap`
- `flatMap` $\sim$ `apply`

Monads und Implicits

- Scala hat *for comprehensions*
- $\rightsquigarrow$ automatische Aufrufe von `map` und `flatMap`

Erkenntnis:

- `map` $\sim$ `fmap`
- `flatMap` $\sim$ `apply`

Monads und Implicits

Situation: `map` und `flatMap` sind in einem separaten Objekt definiert.

Lösung: Anbieten einer impliziten Konversion

Monads und Implicits

```
class MonadWrapper[M[_], A](val value: M[A]) {
    def map[B](f: A => B)
              (implicit t: Functor[M]): M[B] =
        t.fmap(value, f)

    def flatMap[B](f: A => M[B])
                  (implicit m: Monad[M]): M[B] =
        m.bind(value, f)
}

implicit def pimp[M[_], A](value: M[A]) =
    new MonadWrapper[M, A](value)
```

Monads und Implicits

Anwendung

```
def log[A](a: A, l: String) = Computation(List(l), a)

def func(x: Int) =
    for {
        a <- log(x + 3, "adding to " + x)
        b <- log(a * 2, "multiplying 2")
    } yield b

> func(3)
Computation(List("adding to "...), 12)
```

Monaden gegen Seiteneffekte

Q: What does a Scala Developer do to avoid the side effects of a good whiskey?
A: Put it in a monad and flatMap that sh*t.

– *"The infamous flatMap joke"*

Monaden in der Praxis

Welche Monaden gibt es noch?

- `Option[T]`
- `Traversable[T]`
- `Promise[T]`
- `IO[T]`
- `{ type L[A] = T => A }#L`

Monaden in der Praxis

Welche Monaden gibt es noch?

- `Option[T]`
- `Traversable[T]`
- `Promise[T]`
- `IO[T]`
- `{ type L[A] = T => A }#L`

Monaden in der Praxis

Welche Monaden gibt es noch?

- `Option[T]`
- `Traversable[T]`
- `Promise[T]`
- `IO[T]`
- `{ type L[A] = T => A }#L`

Option

„typsicheres `null`“

- `None`: kein Wert vorhanden
- `Some(x)`: Wert `x` vorhanden

Option

Wie arbeitet man mit `Option`?

```
Option[T] => Boolean
Option[T] => T
```

Option

Wie arbeitet man mit `Option`?

```
Option[T] => Boolean
Option[T] => T
```

oder

```
Option[T] => (T => S) => S => S
```

Option

Erkenntnis: Typannotationen sind wertvolle Dokumentation

Option

Erkenntnis: Typannotationen sind wertvolle Dokumentation

Option als Monade

Idee: Verkettung von Berechnungen, die fehlschlagen können

Demo

Validation

- Compiler zwingt nicht zur Exception-Behandlung
- $\rightsquigarrow$ wird in wenigen Fällen richtig gemacht
- Fehlercode als Rückgabewert lange Zeit verschrien
- besser: Fehlerbehandlung durch Typen unterstützen

Validation

- Compiler zwingt nicht zur Exception-Behandlung
- $\rightsquigarrow$ wird in wenigen Fällen richtig gemacht
- Fehlercode als Rückgabewert lange Zeit verschrien
- besser: Fehlerbehandlung durch Typen unterstützen

Validation

Definition

- `Success(a)`: Wert `a` wurde errechnet
- `Failure(e)`: Fehler `e` aufgetreten

Validation als Monade

```
def flatMap(f: A => Validation[E, B]) = this match {
    case Success(a) => f(a)
    case Failure(e) => Failure(e)
}
```

Demo

Zusammenfassung

1. Scalas Typsystem ist dank Funktionsobjekten und Kinds mächtig genug, um Typklassen ausdrücken zu können.
2. Typklassen erlauben mächtige Abstraktionen, mittels denen man häufig verwendete Funktionen vollkommen generisch schreiben kann.
3. Je mehr Information in den Typen steckt, desto besser.

Ausblick

- Kategorien
- Monad Transformers
- Comonaden
- ...

Fragen?

http://www.lars-hupel.de
http://gplus.to/larsrh

Nächster Vortrag:

Schwarze Magie

Scalas Typsystem ausgenutzt

S32, Donnerstag, 11:20