3.– 6. September 2012
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Gang of Many

Funktionale Entwurfsmuster in Scala

Lars Hupel

Ent·wurfs·mus·ter, das: bewährte Lösungsschablone für wiederkehrende Probleme in der Softwarearchitektur und Softwareentwicklung

Fahrplan

Scala, eine hybride Sprache

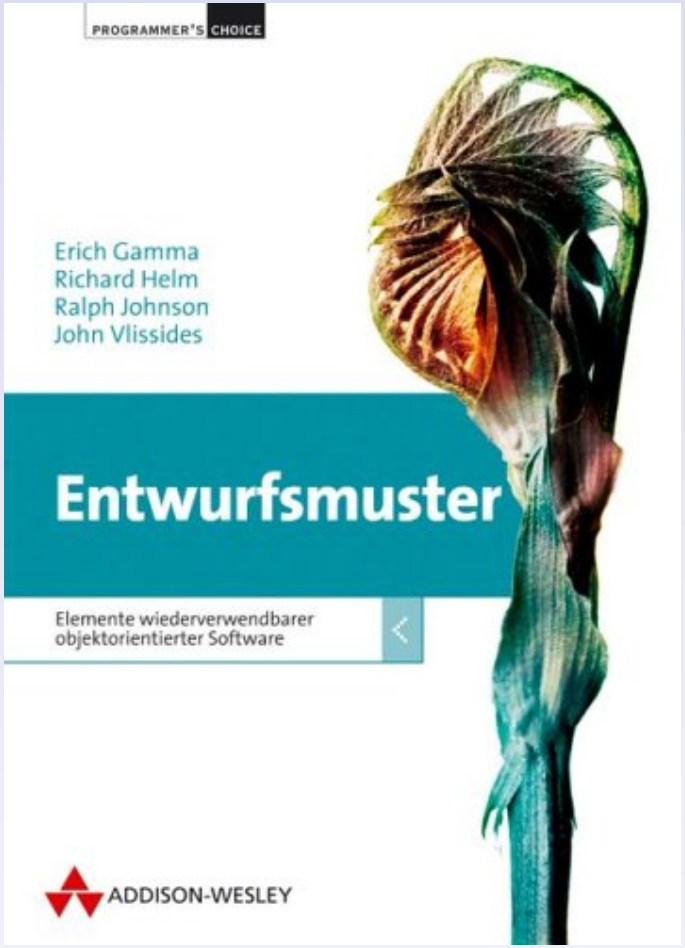
Objektorientierte Entwurfsmuster

- Factory
- Prototype
- Adapter
- Iterator

Funktionale Entwurfsmuster

- Dependency Injection
- Tail-Rekursion

Gang of Four


PROGRAMMER'S CHOICE
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Entwurfsmuster
Elemente wiederverwendbarer objektorientierter Software
ADDISON-WESLEY

Scalas Typsystem

Javas Objektorientierung +

- erweiterte Typinferenz
- Singletons
- implizite Konversionen und Parameter
- ...

⇝ OO-Muster können übernommen werden

Scalas Typsystem

Javas Objektorientierung +

- erweiterte Typinferenz
- Singletons
- implizite Konversionen und Parameter
- ...

$\rightsquigarrow$ OO-Muster können übernommen werden

Summierung à la Scala

```
var total = 0
for (doc <- docs) {
  total += doc.length
}
total
```

Summierung à la Scala

```
var total = 0
docs.foreach(doc => total += doc.length)
total
```

Summierung à la Scala

```
docs.foldLeft(0)((accum, current) =>
  accum + current.length
)
```

Summierung à la Scala

```
docs.foldLeft(0)(_ + _.length)
```

Summierung à la Scala

```
(0 /: docs)(_ + _.length)
```

Summierung à la Scala

```
docs.map(_.length).sum
```

Summierung à la Scala

```
docs.map(_.length).sum
```

```
SELECT SUM(docs.length) FROM docs
```

Summierung à la Scala

```
docs.foldMap(_.length)
```

“Good mathematicians see analogies. Great mathematicians see analogies between analogies.”

– *Stefan Banach*

Factory

Problem

Abstrakte Objekterzeugung

Eigentliches Problem

Abstraktion über Konstruktoren unmöglich

OO-Lösung

Interface mit entsprechenden Signaturen

Factory

Problem

Abstrakte Objekterzeugung

Eigentliches Problem

Abstraktion über Konstruktoren unmöglich

OO-Lösung

Interface mit entsprechenden Signaturen

Factory

Problem

Abstrakte Objekterzeugung

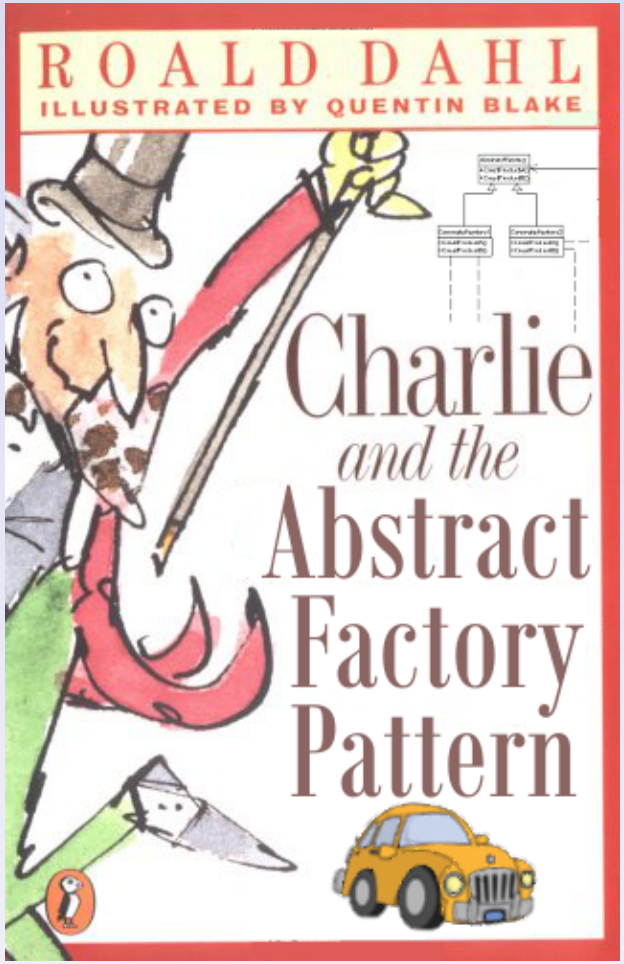
Eigentliches Problem

Abstraktion über Konstruktoren unmöglich

OO-Lösung

Interface mit entsprechenden Signaturen

Factory



https://plus.google.com/115452074099217285141/posts/gWh4YZuQeMK

Factory in Scala

```
type FictionFactory[T <: Fiction] =
  (Int, String) => T

val sciFi: FictionFactory[SciFi] =
  new SciFi(true, _, _)
```

Factory in Scala

Problem

Wie organisiert man die Factories?

Lösungsidee

Als Implicits. Aber wo? Zentrales Objekt?

Factory in Scala

Problem

Wie organisiert man die Factories?

Lösungsidee

Als Implicits. Aber wo? Zentrales Objekt?

Exkurs: Implicit Scope

oder: Wo sucht der Compiler nach Implicits?

Scala Language Specification §7.2

1. importierte Bezeichner
2. aus den Companions von:
   - bei einem Typen `T[A, B, C, ...]`: `T`, `A`, `B`, ...
   - bei einem Typen `A with B with ...`: `A`, `B`, ...

Factory in Scala

Problem

Wie organisiert man die Factories?

Lösung

Als Implicits in Companions.

```
object SciFi {
  implicit def factory: FictionFactory[SciFi] = // ...
}

implicitly[FictionFactory[SciFi]]
```

Exkurs: Context Bounds

Scala vor 2.8

```
def apply[T](implicit ev: Ordering[T]) = // ...
```

Scala seit 2.8

```
def apply[T : Ordering] = // ...
```

Factory in Scala

Verwendung

Java

```
<T> String doSomething(
  Factory<T> factory, int nTimes
);
```

Scala

```
def doSomething[T : Factory](nTimes: Int)
```

Factory in Scala

Evaluation

Vorteile

- Factory = Funktion
- beliebig zusammensetzbar
- weniger Boilerplate

Prototype

Problem

Erzeugung von gleichartigen Objekten

OO-Lösung

```
Object clone();
```

Prototype

Problem

Erzeugung von gleichartigen Objekten

OO-Lösung

```
Object clone();
```

Prototype

`clone` hat in klassischer OOP-Modellierung mehrere Probleme:

- “self type”
- mutable state

Prototype

`clone` hat in klassischer OOP-Modellierung mehrere Probleme:

- “self type”
- mutable state

Exkurs: Self types

Naive Lösung

```
trait Item {
  def copy: Item
}

class FluxCapacitor extends Item {
  def copy = new FluxCapacitor
}

def mkCopy[T <: Item](item: T) = item.copy
```

mkCopy hat Typ Item

Exkurs: Self types

Naive Lösung

```
trait Item {
  def copy: Item
}

class FluxCapacitor extends Item {
  def copy = new FluxCapacitor
}

def mkCopy[T <: Item](item: T) = item.copy
```

mkCopy hat Typ Item

Exkurs: Self types

Lösung mit Dependent Types

```
trait Item {
  def copy: this.type
}

class FluxCapacitor extends Item {
  def copy = new FluxCapacitor
}

def mkCopy[T <: Item](item: T) = item.copy
```

kompiliert nicht

Exkurs: Self types

Lösung mit Dependent Types

```
trait Item {
  def copy: this.type
}

class FluxCapacitor extends Item {
  def copy = new FluxCapacitor
}

def mkCopy[T <: Item](item: T) = item.copy
```

kompiliert nicht

Exkurs: Self types

Gängige Lösung

```
trait Item {
  type Self <: Item
  def copy: Self
}
class FluxCapacitor extends Item {
  type Self = FluxCapacitor
  def copy = new FluxCapacitor
}
def mkCopy[T <: Item](item: T) = item.copy
```

besser, aber immer noch Boilerplate

Exkurs: Self types

Gängige Lösung

```
trait Item {
  type Self <: Item
  def copy: Self
}
class FluxCapacitor extends Item {
  type Self = FluxCapacitor
  def copy = new FluxCapacitor
}
def mkCopy[T <: Item](item: T) = item.copy
```

besser, aber immer noch Boilerplate

Exkurs: Self types

```
trait GenSeqViewLike[
  +A, +Coll,
  +This <: GenSeqView[A, Coll] with
    GenSeqViewLike[A, Coll, This]
] extends GenSeq[A] with GenSeqLike[A, This]
  with GenIterableView[A, Coll]
  with GenIterableViewLike[A, Coll, This]
```

Exkurs: Self types

```
trait GenSeq
  +A, +Coll,
  +This <: G
    GenSeqVi
] extends Ge                    This]
  with GenIt
  with GenIt                    s]
```

Prototype

`clone` hat in klassischer OOP-Modellierung mehrere Probleme:

- “self type”
- mutable state

Prototype

`clone` hat in klassischer OOP-Modellierung mehrere Probleme:

- “self type”
- mutable state

Cloning mit Mutable State

Was ist eine Kopie?

Kopiertes Objekt ...

- referenziert die selben Objekte?
- referenziert Kopien auf die Objekte?
- ist unabhängig?
- hat noch den gleichen State?

Cloning mit Mutable State

Was ist eine Kopie?

Kopiertes Objekt ...

- referenziert die selben Objekte?
- referenziert Kopien auf die Objekte?
- ist unabhängig?
- hat noch den gleichen State?

Cloning mit Mutable State

Was ist eine Kopie?

Kopiertes Objekt ...

- referenziert die selben Objekte?
- referenziert Kopien auf die Objekte?
- ist unabhängig?
- hat noch den gleichen State?

Cloning mit Mutable State

Was ist eine Kopie?

Kopiertes Objekt ...

- referenziert die selben Objekte?
- referenziert Kopien auf die Objekte?
- ist unabhängig?
- hat noch den gleichen State?

Cloning mit Mutable State

Was ist eine Kopie?

Kopiertes Objekt ...

- referenziert die selben Objekte?
- referenziert Kopien auf die Objekte?
- ist unabhängig?
- hat noch den gleichen State?

Cloning mit Mutable State

Workflow

1. Objekt kopieren
2. State ändern

Warum nicht beides kombinieren?

Cloning mit Mutable State

Workflow

1. Objekt kopieren
2. State ändern

Warum nicht beides kombinieren?

Prototype in Scala

Daten als Case Classes

```
case class Person(
  name: String,
  address: String,
  age: Int
)

val lars = Person("Lars", "Munich", 21)
lars.copy(age = 22)
```

Prototype in Scala

Vorteile

- Immutable
- Syntaxzucker

Nachteile

- nicht zusammensetzbar für verschachtelte Strukturen

⇝ skaliert nicht

Prototype in Scala

Vorteile

- Immutable
- Syntaxzucker

Nachteile

- nicht zusammensetzbar für verschachtelte Strukturen

$\rightsquigarrow$ skaliert nicht

Prototype in Scala

Verallgemeinerung des Problems

- gegeben: eine Klasse R (“Record”)
- Ziel: ein Wert F (“Field”), der in R enthalten ist

Operationen

```
def get(object: R): F
def set(object: R, newValue: F): R
```

Prototype in Scala

Verallgemeinerung des Problems

- gegeben: eine Klasse R (“Record”)
- Ziel: ein Wert F (“Field”), der in R enthalten ist

Operationen

```
def get(object: R): F
def set(object: R, newValue: F): R
```

Prototype in Scala

Verallgemeinerung des Problems

- gegeben: eine Klasse R (“Record”)
- Ziel: ein Wert F (“Field”), der in R enthalten ist

Operationen

```
def get(object: R): F
def set(object: R, newValue: F): R
```

Lenses in Scala

```
case class Lens[R, F](
  get: R => F,
  set: (R, F) => F
)
```

Lenses in Scala

```
case class Lens[R, F](
  get: R => F,
  set: (R, F) => F
)
```

Benutzung

```
object Person {
  val name: Lens[Person, Name] = // ...
  val age: Lens[Person, Int] = // ...
}

Person.age.set(lars, 22)
```

Lenses in Scala

```
case class Lens[R, F](
  get: R => F,
  set: (R, F) => F
)
```

Erzeugung

- Deklaration im Companion
- aber: immer noch Boilerplate
- automatische Generierung?

Lenses in Scala

Automatische Generierung

Demo

Lenses in Scala

Literatur

Tony Morris: Asymmetric Lenses in Scala, `http://days2012.scala-lang.org/sites/days2012/files/morris_lenses.pdf`

Adapter

Problem

Integration und Zusammenarbeit von verschiedenen Bibliotheken

Eigentliches Problem

Nachrüsten von Operationen auf Klassen schwierig

OO-Lösung(en)

- Composition (Wrapping)
- (Inheritance)

Adapter

Problem

Integration und Zusammenarbeit von verschiedenen Bibliotheken

Eigentliches Problem

Nachrüsten von Operationen auf Klassen schwierig

OO-Lösung(en)

- Composition (Wrapping)
- (Inheritance)

Adapter

Problem

Integration und Zusammenarbeit von verschiedenen Bibliotheken

Eigentliches Problem

Nachrüsten von Operationen auf Klassen schwierig

OO-Lösung(en)

- Composition (Wrapping)
- (Inheritance)

Adapter in Scala

- leichtgewichtige Wrapper mit Case Classes

```
case class JavaList[T](
  underlying: java.util.LinkedList[T]
) extends Seq[A] {
  // ...
}
```

- automatisches Wrapping mit Implicit Conversions

Type Classes

Andere Möglichkeit: über die Operationen abstrahieren

$\rightsquigarrow$ Typklassen

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

Type Classes

Andere Möglichkeit: über die Operationen abstrahieren

$\rightsquigarrow$ Typklassen

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

Type Classes

Andere Möglichkeit: über die Operationen abstrahieren

$\rightsquigarrow$ Typklassen

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

Type Classes

Andere Möglichkeit: über die Operationen abstrahieren

$\rightsquigarrow$ Typklassen

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

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

– *Paul Hudak*

Type Classes in Scala

Beispiel: Numeric

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

Type Classes in Scala

Beispiel: Numeric

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

Type Classes in Scala

Beispiel: Numeric

```
def solveEquations[T : Numeric](
  system: Array[Array[T]]
): Option[Array[T]] = // ...
```

Type Classes in Scala

Vorteile

- verringert Boilerplate
- Decoupling
- nachträgliche Integration
- kein “self type”-Problem

Nachteile

- Runtime-Overhead

Iterator

Problem

Sequentieller Zugriff auf Container-artige Objekte

OO-Lösung

Iterator-Objekt mit den Operationen

- `next`
- `get`
- `prev`
- ...

Iterator in Scala

Iterator kann ersetzt werden durch:

- `map`
- `foldLeft/foldRight`
- `flatMap`
- ...

Iterator in Scala

http://www.gedraengel.de/alter-hut/

Iterator in Scala

http://golem.ph.utexas.edu/category/2012/01/vorsicht_funktor.html

Applicative

Gängige Situation: „annotierter“ Wert

- `Future`
- `Callable`
- Wert oder `null`
- Wert oder Exception
- nichtdeterministisches Ergebnis

⇝ Wert mit Kontext

Applicative

Gängige Situation: „annotierter“ Wert

- `Future`
- `Callable`
- `Option`
- Wert oder Exception
- nichtdeterministisches Ergebnis

⇝ Wert mit Kontext

Applicative

Gängige Situation: „annotierter“ Wert

- `Future`
- `Callable`
- `Option`
- `Either/Validation/Try`
- nichtdeterministisches Ergebnis

⇝ Wert mit Kontext

Applicative

Gängige Situation: „annotierter“ Wert

- `Future`
- `Callable`
- `Option`
- `Either/Validation/Try`
- `List`

⇝ Wert mit Kontext

Applicative

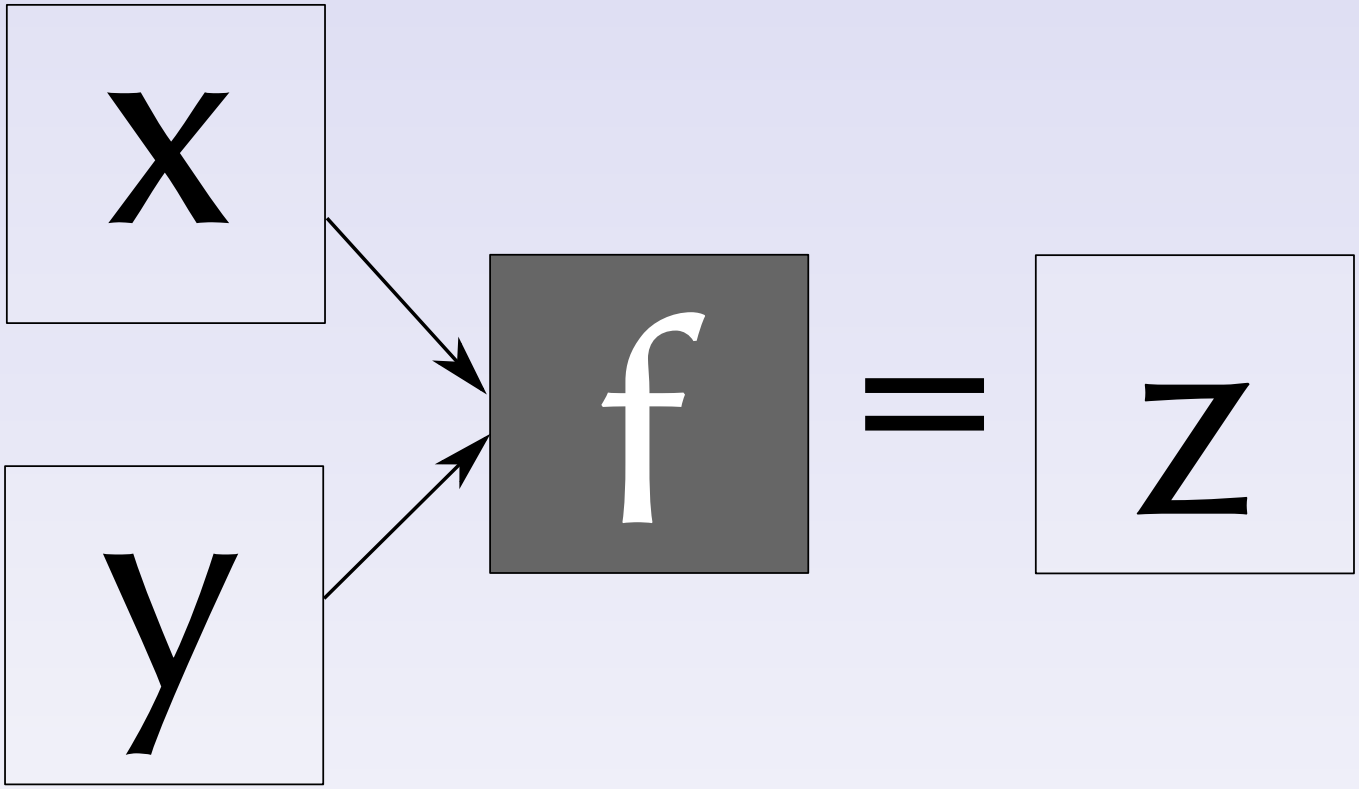
Gängige Situation: „annotierter“ Wert

- `Future`
- `Callable`
- `Option`
- `Either/Validation/Try`
- `List`

⇝ Wert mit Kontext

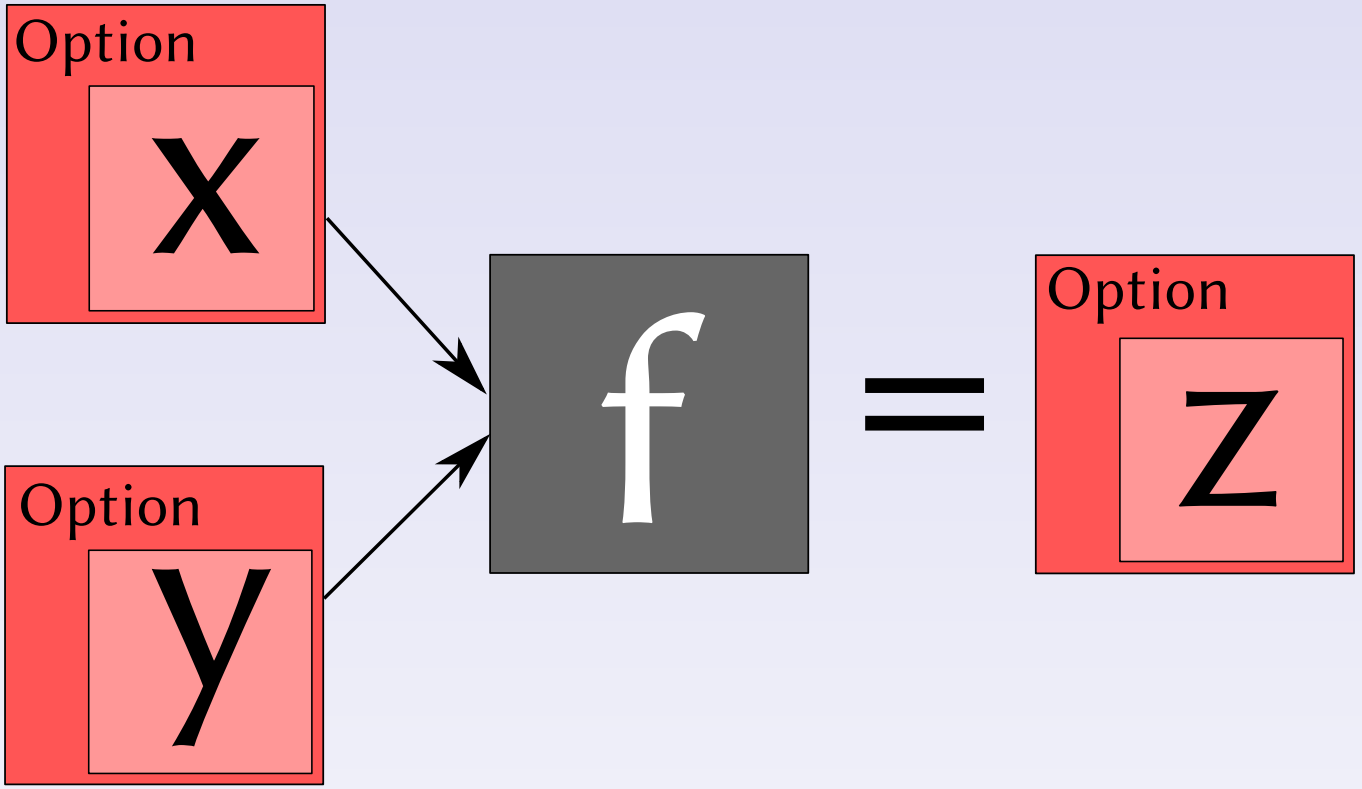
Applicative

Funktionsanwendung



Applicative

Funktionsanwendung



Traverse

Aufgabe

Eingabe: Liste von Werten, die `null` sein können

Ausgabe: Liste der gleichen Werte, falls alle nicht-`null` sind

Traverse

Aufgabe

Eingabe: Liste von Werten, die `null` sein können

Ausgabe: Liste der gleichen Werte, falls alle nicht-`null` sind

Lösung

```
list.traverse(Option(_))
```

Traverse

Demo

Traverse

Literator

Eric Torreborre: The Essence of the Iterator Pattern, `http://etorreborre.blogspot.de/2011/06/essence-of-iterator-pattern.html`

Jeremy Gibbons, Bruno C.d.S. Oliveira: The Essence of the Iterator Pattern, `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf`

Dependency Injection

CDI, Spring, Guice, ...

... verlassen den Sprachumfang

Dependency Injection

CDI, Spring, Guice, ...

... verlassen den Sprachumfang

Dependency Injection in Scala

1. Cake Pattern
2. Reader Monad

Dependency Injection in Scala

1. Cake Pattern
2. Reader Monad

Cake Pattern

Grundidee

- Aufteilung des Codes in mehrere Komponenten
- jede Komponente hat Abhängigkeiten zu anderen Komponenten
- zyklische Abhängigkeiten erlaubt

Cake Pattern

Beispiel

```
trait Configurations {

  val configuration: Configuration

  class Configuration(file: String) {
    def jdbcSource: String = // ...
    // ...
  }

}
```

Cake Pattern

Beispiel

```
trait Connections {
  this: Configurations =>

  val connection: Connection

  class Connection(cacheSize: Int) {
    open(configuration.jdbcSource)
    // ...
  }

}
```

Cake Pattern

Beispiel

```
object Production
  extends Configurations
  with Connections {

  val configuration =
    new Configuration("conf.prod")

  val connection =
    new Connection(1024)

}
```

Cake Pattern

Beispiel

```
object Testing
  extends Configurations
  with Connections {

  val configuration =
    new Configuration("conf.test")

  val connection =
    new Connection(0)

}
```

Cake Pattern

Vorteile

- läuft innerhalb der Sprache ab
- Compiler prüft Dependencies
- keine externe Konfiguration
- sehr flexibel, weil voller Sprachumfang verfügbar

Nachteile

- erfordert Dependent Types
- kompiliert u.U. langsam

Dependency Injection in Scala

1. Cake Pattern
2. Reader Monad

Dependency Injection in Scala

1. Cake Pattern
2. Reader Monad

Reader Monad

Problem

Konfiguration muss bis in die untersten Layer durchgereicht werden

Lösungsansatz

Funktion, die Konfiguration erwartet

Currying

```
def computeTotal(items: List[Item], t: Tariff)
  : Currency
```

Currying

```
def computeTotal
  (items: List[Item])
  (t: Tariff): Currency
```

Currying

```
type TariffReader[A] = Tariff => A

def computeTotal(items: List[Item])
  : TariffReader[Currency] = { t =>
  // ...
}
```

Reader Monad

Verwendung

```
for {
  total <- computeTotal(items)
  discounted <- applyDiscount(total, cust)
} yield
  eligiblePayments(discounted, cust)
```

Reader Monad

- `TariffReader` verhält sich monadisch

⇝ zusammensetzbar

- Struktur des Programms ist unabhängig vom Kontext
- “dead simple”

Iteration vs. Rekursion

Iteration böse

Iteration vs. Rekursion

Rekursion gut

Rekursion

```
def sum(n: Int): Int =
  if (n > 0)
    n + sum(n - 1)
  else
    0
```

Rekursion

```
def sum(n: Int): Int =
  if (n >
    n + su
  else
    0
```

Rekursion

```scala
def sum(n: Int) = {
  def aux(curr: Int, acc: Int): Int =
    if (curr > 0)
      aux(curr - 1, acc + curr)
    else
      0

  aux(n, 0)
}
```

Fragen?

larsrh.github.com
lars.hupel@tum.de