MATHEMA campus 2014

Jetzt funkt's!

Funktionale Programmierung für Anfänger

Nicole Rauch

msgGillardon AG

Funktional, das ist doch nur für Esoteriker?!

Funktional, das ist doch nur für Esoteriker?!

- ABN AMRO Amsterdam *Risikoanalysen Investmentbanking*
- AT&T *Netzwerksicherheit: Verarbeitung von Internet-Missbrauchsmeldungen*
- Bank of America Merril Lynch
  Backend: Laden & Transformieren von Daten
- Barclays Capital Quantitative Analytics Group
  Mathematische Modellierung von Derivaten
- Bluespec, Inc. *Modellierung & Verifikation integrierter Schaltkreise*
- Credit Suisse *Prüfen und Bearbeiten von Spreadsheets*
- Deutsche Bank Equity Proprietary Trading, Directional Credit Trading *Gesamte Software-Infrastruktur*
- Facebook *Interne Tools*
- Factis Research, Freiburg *Mobil-Lösungen (Backend)*
- fortytools gmbh *Webbasierte Produktivitätstools - REST-Backend*
- Functor AB, Stockholm *Statische Codeanalyse*

Funktional, das ist doch nur für Esoteriker?!

- Galois, Inc *Security, Informationssicherheit, Kryptographie*
- Google *Interne Projekte*
- iPwn Studios *Spieleengine*
- JanRain *Netzwerk- und Web-Software*
- MITRE *Analyse kryptographischer Protokolle*
- New York Times *Bildverarbeitung für die New York Fashion Week*
- NVIDIA *In-House Tools*
- Parallel Scientific *Hochskalierbares Cluster-Verwaltungssystem*
- Sankel Software *CAD/CAM, Spiele, Computeranimation*
- Silk, Amsterdam *Filtern und Visualisieren großer Datenmengen*
- Skedge Me *Online-Terminvereinbarungen*
- Standard Chartered *Bankensoftware*
- Starling Software, Tokio *Automatisiertes Optionshandelssystem*
- Suite Solutions *Verwaltung technischer Dokumentationen*

(Quelle: `http://www.haskell.org/haskellwiki/Haskell_in_industry`)

Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

Lisp

Scala

Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

Lisp

Scala

(JavaScript)

Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

Lisp

Scala

(JavaScript)

Was ist denn an funktionaler Programmierung so besonders?

Was ist denn an funktionaler Programmierung so besonders?

Funktionen sind „first order citizens“

Was ist denn an funktionaler Programmierung so besonders?

Funktionen sind „first order citizens“

Mit Funktionen kann man dasselbe machen wie mit Strings oder Zahlen

Funktionen können Variablen zugewiesen werden

Funktionen können Variablen zugewiesen werden

JavaScript:

```
function times(x, y) { return x * y; }

var timesVar = times;

timesVar(3, 5) === 15;
```

Funktionen können Variablen zugewiesen werden

JavaScript:

```
function times(x, y) { return x * y; }

var timesVar = times;

timesVar(3, 5) === 15;
```

Haskell:

```
times x y = x * y

timesVar = times

timesVar 3 5 == 15
```

Funktionen können als Funktionsparameter übergeben werden

Funktionen können als Funktionsparameter übergeben werden

JavaScript:

```
function wendeAn(func, arg) { return func(arg); }

function times3(y) { return 3 * y; };

wendeAn(times3, 5) === 15;
```

Funktionen können als Funktionsparameter übergeben werden

JavaScript:

```
function wendeAn(func, arg) { return func(arg); }

function times3(y) { return 3 * y; };

wendeAn(times3, 5) === 15;
```

Haskell:

```
wendeAn func arg = func arg

wendeAn (\ x -> 3 * x) 5 == 15
```

Funktionen können von Funktionen zurückgegeben werden

Funktionen können von Funktionen zurückgegeben werden

JavaScript:

```
function times(x) { return function (y) { return x * y; }; }

times(3)(5) === 15;
```

Funktionen können von Funktionen zurückgegeben werden

JavaScript:

```
function times(x) { return function (y) { return x * y; }; }

times(3)(5) === 15;
```

Haskell:

```
times x = (\y -> x * y)

times 3 5 == 15
```

Komisch, oder?

JavaScript: Zwei verschiedene Aufrufe

```
function times(x, y) { return x * y; }
times(3, 5) === 15;

function times(x) { return function (y) { return x * y; }; }
times(3)(5) === 15;
```

Haskell: Zweimal derselbe Aufruf

```
times x y = x * y
times 3 5 == 15

times x = (\y -> x * y)
times 3 5 == 15
```

Currying! (oder auch Schönfinkeln)

Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

Denn: Funktionen haben immer nur ein Argument

Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

Denn: Funktionen haben immer nur ein Argument

Nutzen: Partielle Evaluierung:

```
times x y = x * y

times3 = times 3

times3 5 == 15
```

Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer ein Argument!
- Was ist, wenn ich nichts habe?!

Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer ein Argument!
- Was ist, wenn ich nichts habe?!
- `Unit` to the rescue!

Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer ein Argument!
- Was ist, wenn ich nichts habe?!
- `Unit` to the rescue!
- `Unit` ist ein Typ mit nur einem Element

Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer ein Argument!
- Was ist, wenn ich nichts habe?!
- `Unit` to the rescue!
- `Unit` ist ein Typ mit nur einem Element
- Das Element heißt in Haskell `()`

Wichtige Bibliotheksfunktionen: filter

- filter oder auch select

Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

JavaScript: (mit freundlicher Unterstützung der lodash-Bibliothek)

```
var result = _.filter([1, 2, 3, 4], function (x) { return x % 2 === 0; });

Ergebnis: [2, 4];
```

Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

JavaScript: (mit freundlicher Unterstützung der lodash-Bibliothek)

```
var result = _.filter([1, 2, 3, 4], function (x) { return x % 2 === 0; });

Ergebnis: [2, 4];
```

Haskell:

```
filter (\x -> x `mod` 2 == 0) [1,2,3,4] == [2,4]
```

Wichtige Bibliotheksfunktionen: map

- map oder auch collect

Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

JavaScript:

```
var result = _.map( [1, 2, 3, 4], function (x) { return x + 5; } );

Ergebnis: [6, 7, 8, 9];
```

Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

JavaScript:

```
var result = _.map( [1, 2, 3, 4], function (x) { return x + 5; } );

Ergebnis: [6, 7, 8, 9];
```

Haskell:

```
map (\x -> x + 5) [1,2,3,4] == [6,7,8,9]
```

Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject

Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject
- Nimmt eine Collection und eine Funktion
- Verbindet Startwert und erstes Element (oder zwei Elemente) der Collection mit Hilfe der Funktion zu einem Element
- Verbindet das Ergebnis mit dem nächsten Element der Collection zu einem Element
- Setzt dies für alle Elemente der Collection fort, bis nur noch ein Element übrigbleibt

Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject
- Nimmt eine Collection und eine Funktion
- Verbindet Startwert und erstes Element (oder zwei Elemente) der Collection mit Hilfe der Funktion zu einem Element
- Verbindet das Ergebnis mit dem nächsten Element der Collection zu einem Element
- Setzt dies für alle Elemente der Collection fort, bis nur noch ein Element übrigbleibt

JavaScript: (mit oder ohne Startwert)

```
var result = _.foldl( [2, 3, 4, 5], function (x, y) { return x * y; } );
var result = _.foldl( [2, 3, 4, 5], function (x, y) { return x * y; }, 1 );
result === 120;
```

Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject
- Nimmt eine Collection und eine Funktion
- Verbindet Startwert und erstes Element (oder zwei Elemente) der Collection mit Hilfe der Funktion zu einem Element
- Verbindet das Ergebnis mit dem nächsten Element der Collection zu einem Element
- Setzt dies für alle Elemente der Collection fort, bis nur noch ein Element übrigbleibt

JavaScript: (mit oder ohne Startwert)

```
var result = _.foldl( [2, 3, 4, 5], function (x, y) { return x * y; } );
var result = _.foldl( [2, 3, 4, 5], function (x, y) { return x * y; }, 1 );
result === 120;
```

Haskell (nur mit Startwert):

```
foldl (*) 1 [2,3,4,5] == 120
```

Typinferenz

- Haskell: starkes statisches Typsystem
- Leichtgewichtige Verwendung dank Typinferenz
- Herleitung des allgemeinst möglichen Typs

Typinferenz

- Haskell: starkes statisches Typsystem
- Leichtgewichtige Verwendung dank Typinferenz
- Herleitung des allgemeinst möglichen Typs

```
foldl :: (a -> b -> a) -> a -> [b] -> a
(*) :: Num a => a -> a -> a

foldl (*) 1 [2,3,4,5]
```

Typinferenz

- Haskell: starkes statisches Typsystem
- Leichtgewichtige Verwendung dank Typinferenz
- Herleitung des allgemeinst möglichen Typs

```
foldl :: (a -> b -> a) -> a -> [b] -> a
(*) :: Num a => a -> a -> a

foldl (*) 1 [2,3,4,5]
```

Compilerfehler für:

```
foldl (*) "x" [2,3,4,5]

No instance for (Num [Char]) arising from a use of ‘*’
Possible fix: add an instance declaration for (Num [Char])
```

Eine einfache Berechnung

$$summe = \sum_{i=1}^{10} i^2$$

Eine einfache Berechnung

$$summe = \sum_{i=1}^{10} i^2$$

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

Exkurs: Clean Code

Single Responsibility Principle

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen

Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int summe = 0;
for(int i = 1; i <= 10; i++) {
  summe = summe + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen
- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen
- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
var zahlenfolge = _.range(1, 11);
```

- Quadrieren einer Zahl

- Berechnen der Quadratzahl jeder Zahl in der Folge

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
var zahlenfolge = _.range(1, 11);
```

- Quadrieren einer Zahl

```
var quadriere = function (i) { return i * i; };
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
var zahlenfolge = _.range(1, 11);
```

- Quadrieren einer Zahl

```
var quadriere = function (i) { return i * i; };
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
var quadrierteFolge = _.map(zahlenfolge, quadriere)
```

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
var zahlenfolge = _.range(1, 11);
```

- Quadrieren einer Zahl

```
var quadriere = function (i) { return i * i; };
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
var quadrierteFolge = _.map(zahlenfolge, quadriere)
```

- Addieren zweier Zahlen

```
var addiere = function (summe, summand) { return summe + summand; };
```

- Aufsummieren der Quadratzahlen

Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
var zahlenfolge = _.range(1, 11);
```

- Quadrieren einer Zahl

```
var quadriere = function (i) { return i * i; };
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
var quadrierteFolge = _.map(zahlenfolge, quadriere)
```

- Addieren zweier Zahlen

```
var addiere = function (summe, summand) { return summe + summand; };
```

- Aufsummieren der Quadratzahlen

```
var summe = _.reduce(quadrierteFolge, addiere);
```

Zusammensetzen der Komponenten

JavaScript:

```
var quadriere = function (i) { return i * i; };
var addiere = function (summe, summand) { return summe + summand; };
```

```
_.reduce(_.map(_.range(1, 11), quadriere), addiere) === 385;
```

Zusammensetzen der Komponenten

JavaScript:

```
var quadriere = function (i) { return i * i; };
var addiere = function (summe, summand) { return summe + summand; };
```

```
_.reduce(_.map(_.range(1, 11), quadriere), addiere) === 385;
```

oder

```
_(1).range(11).map(quadriere).reduce(addiere) === 385;
```

Zusammensetzen der Komponenten

JavaScript:

```
var quadriere = function (i) { return i * i; };
var addiere = function (summe, summand) { return summe + summand; };
```

```
_.reduce(_.map(_.range(1, 11), quadriere), addiere) === 385;
```

oder

```
_(1).range(11).map(quadriere).reduce(addiere) === 385;
```

Haskell:

```
foldl (+) 0 (map (\x -> x*x) [1..10]) == 385
```

oder

```
(>.>) x f = f x
[1..10] >.> map (\x -> x*x) >.> foldl (+) 0 == 385
```

Uff!

OK, alle einmal tief durchatmen :-)

Pattern Matching

Fibonacci-Funktion „naiv“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

Pattern Matching

Fibonacci-Funktion „naiv“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

Fibonacci-Funktion mit Pattern Matching:

```
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x
```

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

```
treeSum myTree == 15
```

Fazit

- Funktionale Programmierung ist verbreiteter als man denkt
- Java 8 hat funktionale Neuerungen
- Viele Sprachen bringen funktionale Aspekte oder Zusatzmodule mit

Fazit

- Funktionale Programmierung ist verbreiteter als man denkt
- Java 8 hat funktionale Neuerungen
- Viele Sprachen bringen funktionale Aspekte oder Zusatzmodule mit

Referenzen:

- Funktionale JS-Bibliothek: `http://lodash.com`
- Haskell: `http://www.haskell.org`

Code & Folien auf GitHub:

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

Nicole Rauch

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

Twitter `@NicoleRauch`