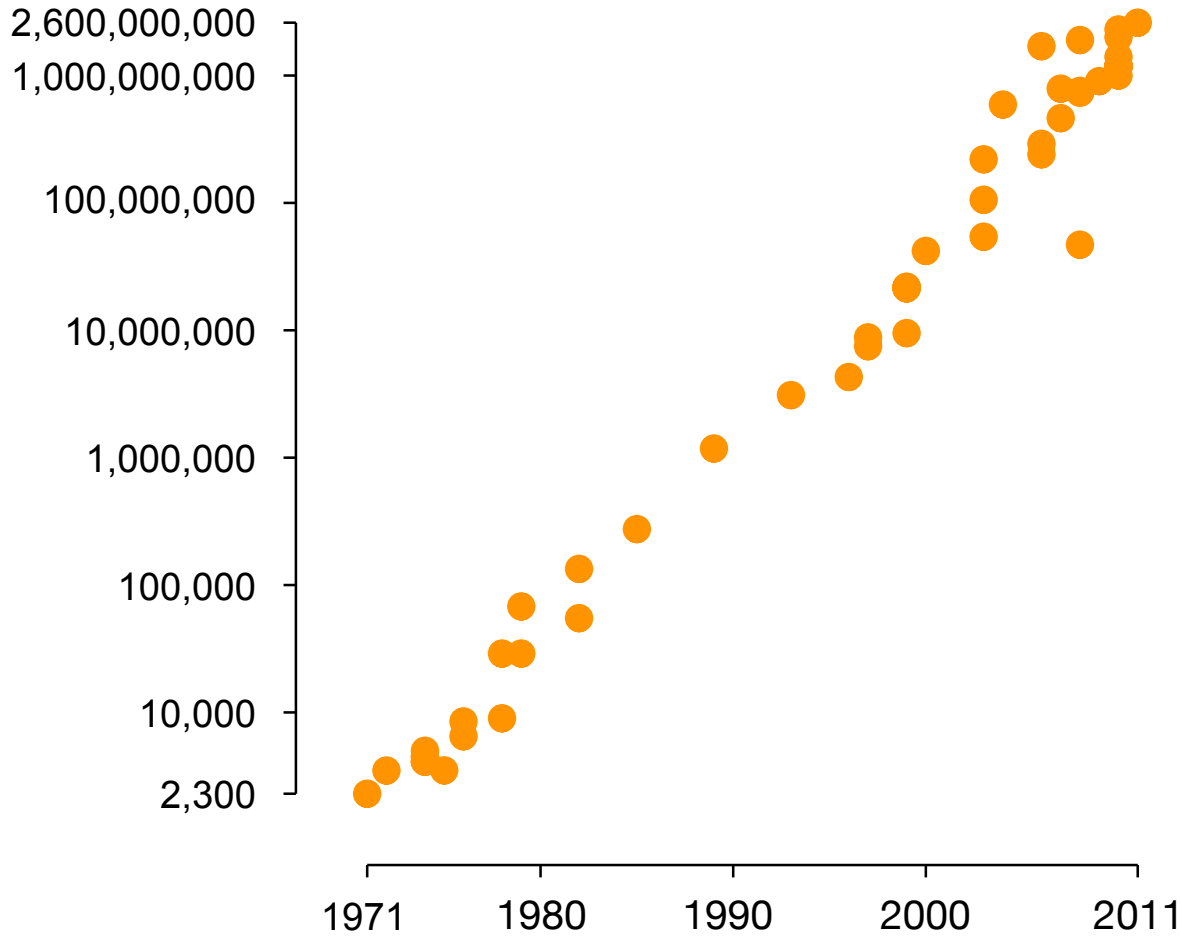
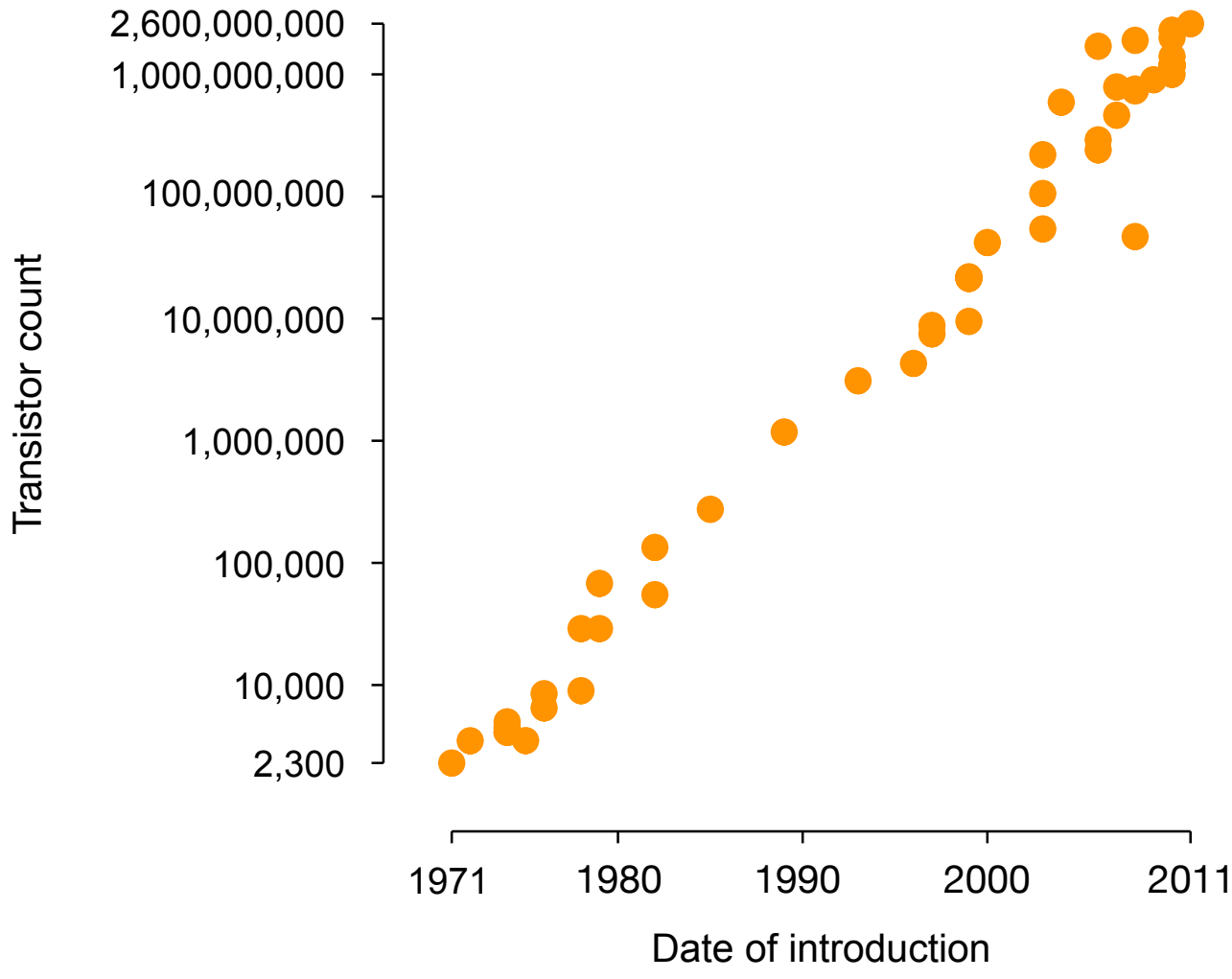
WAS WIRD HIER GEZEIGT?

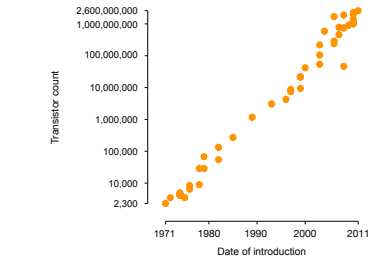


TRANSISTOR COUNTS 1971-2011 & MOORE'S LAW



Funktionale Programierung

Transistor Counts 1971-2011 & Moore’s Law



- relation von transitor anzahl und deren produkt-release
- man sieht augenscheinlich, dass sich die anzahl jeweilig verdoppelt hat
- viele vermuten, dass sich die moore’s law bewahrheitet hat
- man munkelt, dass wir damit nun allmählich an die grenzen gelangt sind

GEAR & GADGETS —

Intel Core i7-7700K Kaby Lake review: Is the desktop CPU dead?

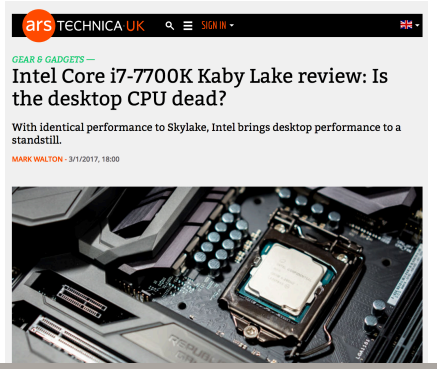
With identical performance to Skylake, Intel brings desktop performance to a standstill.

MARK WALTON - 3/1/2017, 18:00

Funktionale Programierung

└arsTECHNICA UK - 03.01.2017 (1)

ARSTECHNICA UK - 03.01.2017 (1)

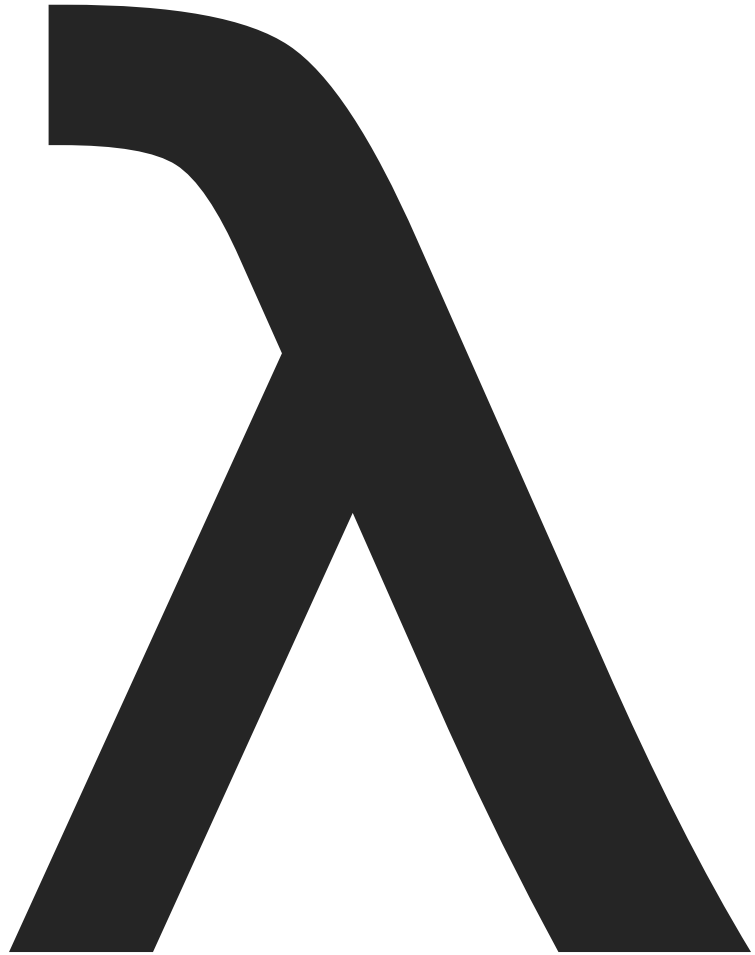


- Man kann interpretieren, dass INTEL keine weitere Leistungssteigerung erziehlen konnte, oder aber auch, dass es gerade im Kontext des BigData in Zukunft wichtiger denn je wird, nebenläufig programmieren zu können.
- deswegen sollten wir meiner Meinung nach die bisherigen Konzepte überdenken

FUNKTIONALE PROGRAMIERUNG

Nebenläufigkeit & Parallelisierung

Seminar, WS2016



Jan-Philipp Willem

Prof. Dr. Sandro Leuchter
Fakultät für Informatik
Hochschule Mannheim

GLIEDERUNG

1. Nebenläufigkeit / Parallelisierung
2. Functional Paradigm 101
3. Elixir
4. Fazit

NEBENLÄUFIGKEIT / PARALLELISIERUNG

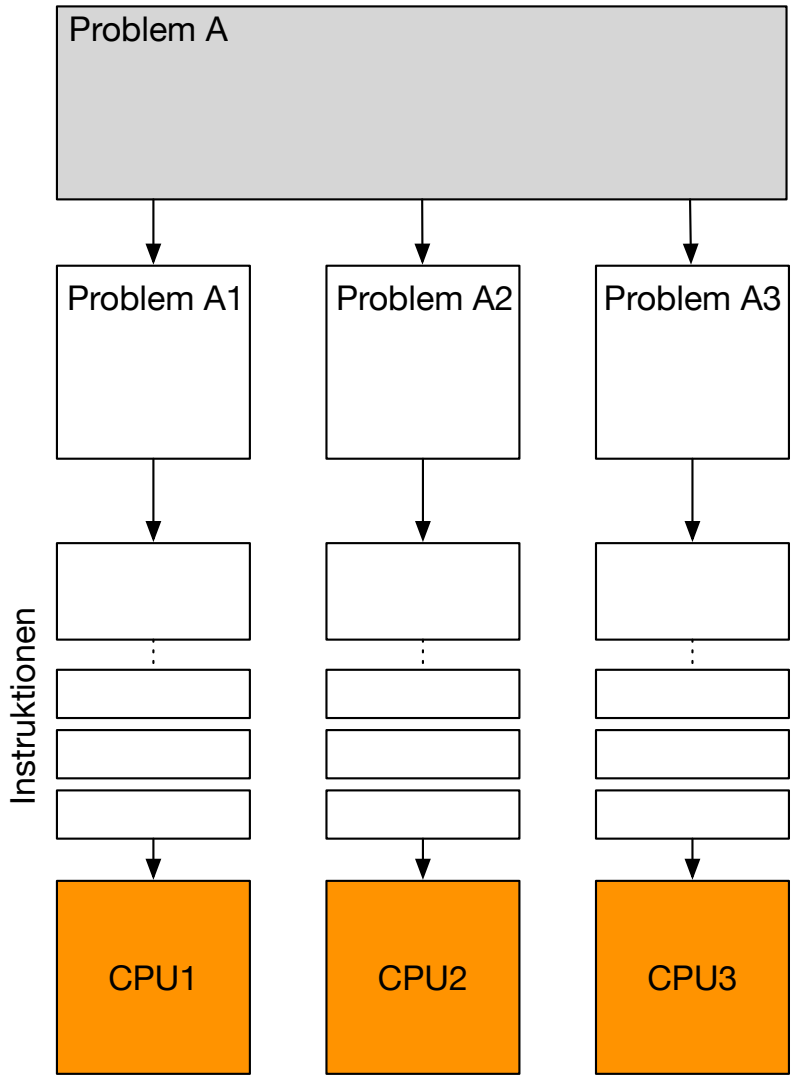
PARALLEL

→ Synonyme:
nebeneinander,
nebenläufig

→ Informatik:
parallel ≠ nebenläufig!

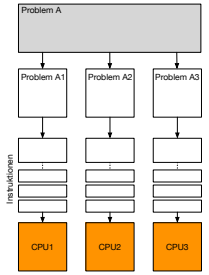
→ „schneller als
sequenzielles Programm,
durch gleichzeitiges
Ausführen von
Anweisungen“

→ Multi-Processing



PARALLEL

→ Synonyme: *nebeneinander, nebenläufig*
→ Informatik: parallel ≠ nebenläufig!
→ „schneller als sequenzielles Programm, durch gleichzeitiges Ausführen von **Anweisungen**“
→ Multi-Processing



- deutscher Begriff schwierig
- Aufteilen des Problems in Teilprobleme
- Teile können, müssen aber nicht zusammengehörend sein

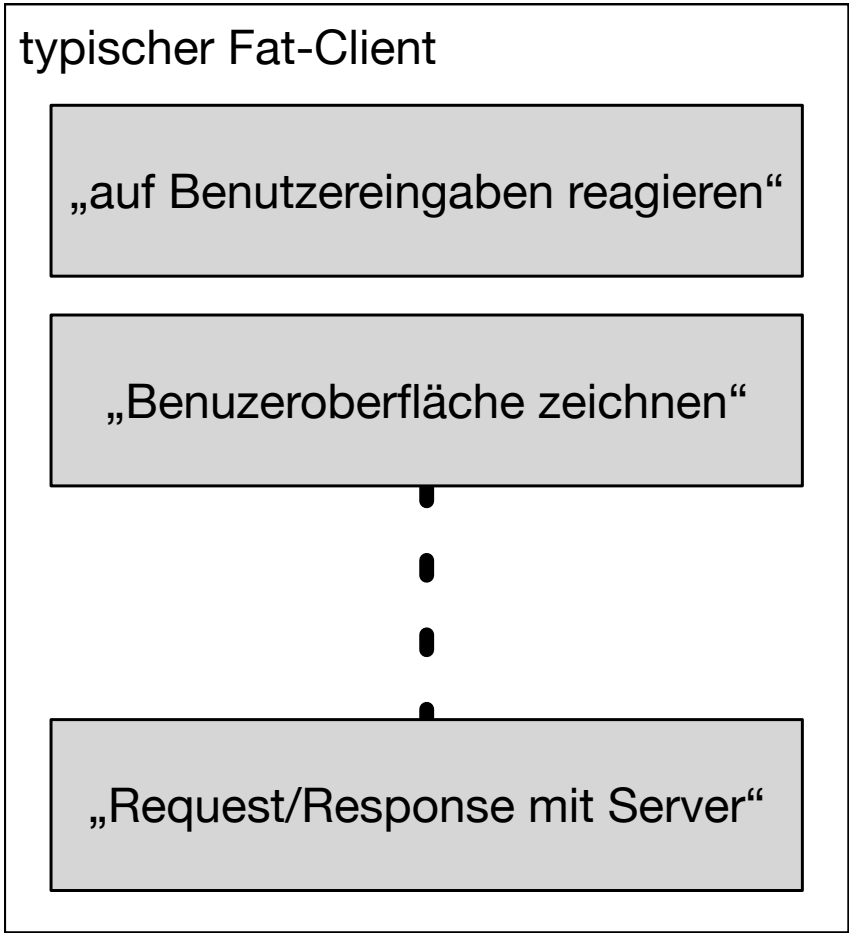
NEBENLÄUFIG

→ concurrent (engl.)

→ „Systeme, welche zur gleichen Zeit mehrere **Aufgaben** haben“

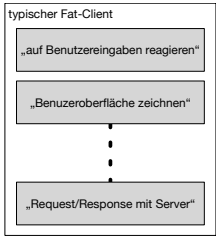
→ muss nicht zwangsläufig parallel sein

→ Multi-Tasking



Funktionale Programierung
Nebenläufigkeit / Parallelisierung
Nebenläufig

- → concurrent (engl.)
- → „Systeme, welche zur gleichen Zeit mehrere **Aufgaben** haben“
- → muss nicht zwangsläufig parallel sein
- → Multi-Tasking



- Fokus liegt auf Arichtektur

ROB PIKE - „CONCURRENCY IS NOT PARALLELISM“ (1)

→ „Concurrency is about **dealing with** lots of things at once.“

→ „Parallelism is about **doing** lots of things at once.“

→ „Concurrency is about structure, parallelism is about execution.“

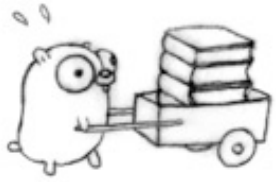
Funktionale Programierung
└Nebenläufigkeit / Parallelisierung
└Rob Pike - „Concurrency Is Not Parallelism“

- Rob Pike, Google, Go-Lang
- interessanter Talk auf Youtube

ROB PIKE - „CONCURRENCY IS NOT PARALLELISM“ (1)

→ „Concurrency is about **dealing with** lots of things at once.“
→ „Parallelism is about **doing** lots of things at once.“
→ „Concurrency is about structure, parallelism is about execution.“

ROB PIKE - „CONCURRENCY IS NOT PARALLELISM“ (2)



→ sequenziell

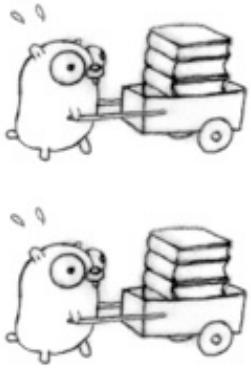
Funktionale Programierung
Nebenläufigkeit / Parallelisierung
Rob Pike - „Concurrency Is Not Parallelism“



→ sequenziell

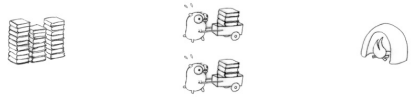
- typischer Aufbau einer App
- Producer/Consumer
- Daten konsolidieren/filtern, Verarbeitung/Transport/Buffer, Weiterverwendung/Ausgabe
- komplett sequenzieller Aufbau, da nur ein worker und alles strikt nacheinander passiert
- problem: ich brauche nur einen gewissen datensatz, dennoch werden alle daten verarbeitet

ROB PIKE - „CONCURRENCY IS NOT PARALLELISM“ (3)



→ parallel

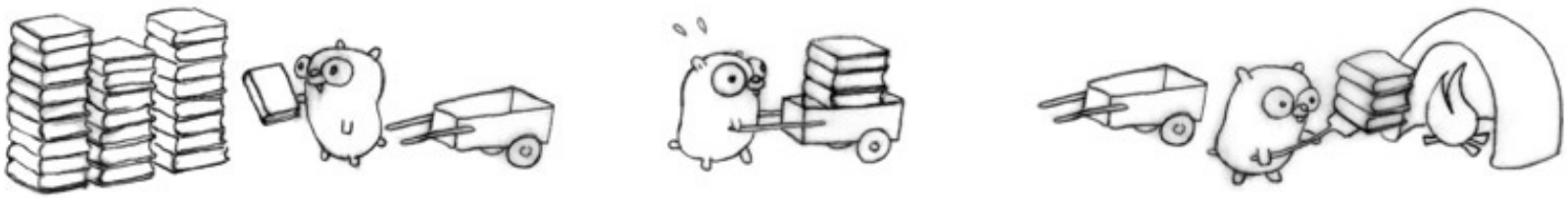
Funktionale Programierung
Nebenläufigkeit / Parallelisierung
Rob Pike - „Concurrency Is Not Parallelism“



→ parallel

- Optimierung einzelner Teile der Applikation
- weitere schritte birgen ähnlich hohen aufwand
- je nach architektur parallelisierung von producer u.U. schwierig

ROB PIKE - „CONCURRENCY IS NOT PARALLELISM“ (4)



→ concurrent

Funktionale Programierung
Nebenläufigkeit / Parallelisierung
Rob Pike - „Concurrency Is Not Parallelism“

→ concurrent

- Nebenläufige Architektur von Anfang an
- d.h. Service-Orientierte Module, die voneinander getrennt funktionieren
- Pro Software-Modul oder Teilsystem eines Moduls kann jeweils ein worker eingesetzt werden.
- architektur kann sehr leicht nach bedarf parallelisiert werden
- nebeneffekt: stabileres System

FUNCTIONAL PARADIGM 101

FUNCTIONAL PARADIGM 101 (1)

→ immutable // mutable

```
 a = 3          a = 3
a += 2          a' = add(a, 2)
a -> 5          a' -> 5
```

→ immutable // mutable

```
 a = 3        a = 3
a += 2        a' = add(a, 2)
a -> 5        a' -> 5
```

1. viele bugs treten durch Veränderung von bestehendem Zustand auf
- in den meisten fällen kann eine Veränderung vermieden werden
- ändert man Zustände, so sollte man besser eine Kopie zurückgeben , anstatt die Referenz zu bearbeiten
- falls gewünscht kann man diese Zustandskopien wie Snapshots in einem Cache betrachten

FUNCTIONAL PARADIGM 101 (2)

→ no side-effects, deterministisches Verhalten

→ „pure“, „data-in, data-out“, EVA

→ functions as first-class citizens

→ lambdas, callbacks

→ no side-effects, deterministisches Verhalten
→ „pure“, „data-in, data-out“, EVA
→ functions as first-class citizens
→ lambdas, callbacks

2. ähnlich sollte eine Funktion auch nur diese eine Sache tun, welche man erwartet -> Side-Effects vermeiden
3. Funktionen können als reine daten-transformationen aufgefasst werden
- man bekommt daten, verarbeitet diese und gibt sie verändert zurück.
4. Funktionen sind selbst auch Typen
- d.h. sie können genauso als parameter einer anderen funktion dienen
- sie werden jedoch erst evaluiert, wenn benötigt
5. lamdas sind funktionen, welche inline definiert werden und somit anonym sind
- sie haben keinen namen, werden aber oft entweder als callback direkt übergeben oder einer variable zugewiesen

FUNCTIONAL PARADIGM 101 (3)

→ reine Funktionale Sprachen

→ reine Funktionale Sprachen

6. Unterscheidung zwischen Sprachen die Unterstützung für fp bieten - und solchen die ausschließlich auf den Konzepten und Prinzipien von Funktionalen Sprachen beruhen
- seiten-effekte sind somit niemals möglich
- meistens strikte Typisierung

ELIXIR

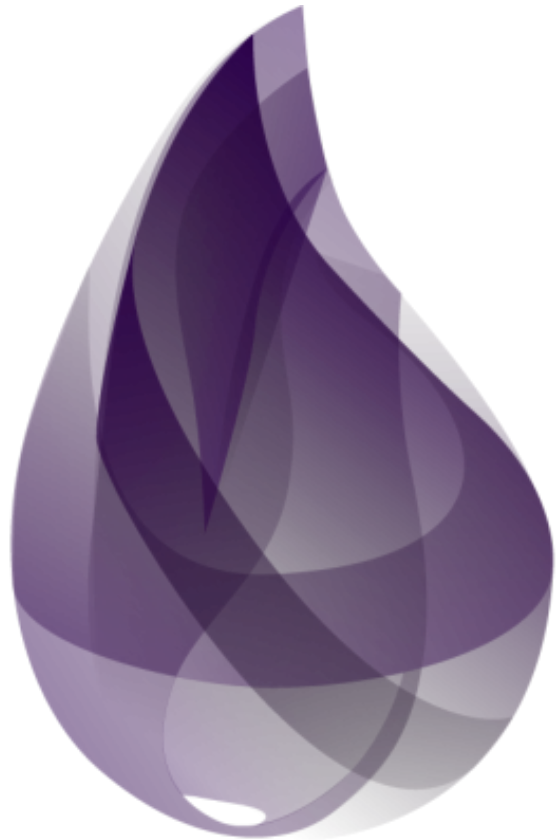
ELIXIR (1)

→ 2011: moderne Variante von Erlang (1987, Ericsson)

→ Beam-VM

→ lightweight Elixir-Processes

→ Shared & Distributed Memory



→ 2011: moderne Variante von Erlang (1987, Ericsson)
→ Beam-VM
→ lightweight Elixir-Processes
→ Shared & Distributed Memory

- funktionale Sprache, welche weitestgehend frei von seiten-effekten ist und nutzt eine dynamische typisierung mit typen-interferenz
- ob sie nun als reine funktionale bezeichnet werden kann streiten sich einige Fronten
- Elixir erweitert Erlang um einige moderne Features wie bspw. metaprogramming und bringt teilweise starke Vereinfachungen bei der Syntax -> ruby
- wird zu erlang bytecode compiliert der in beam-vm läuft
- Nebenläuffigkeit wird mit Elixir-Processes erreicht, welche jedoch um viele Male leichtgewichtiger sind, als System-Prozesse
- je nach anwendungsfall kann man mit elixir sowohl einen gemeinsamen als auch verteilten Zustand nutzen

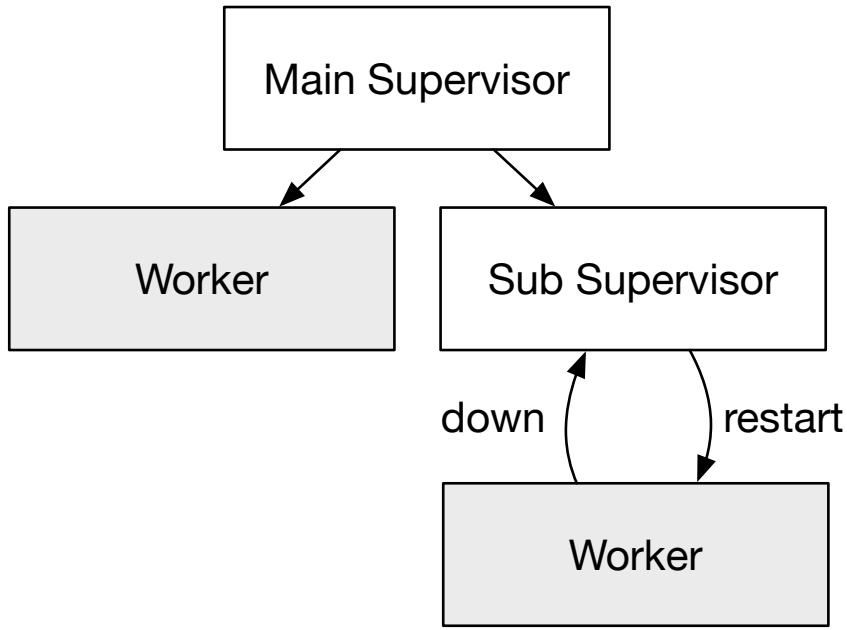
ELIXIR (2)

→ Open Telecom Platform (OTP)

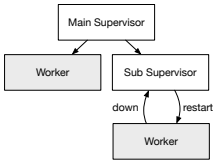
→ Fault-Tolerant

→ „Let it crash“

→ Supervision-Trees



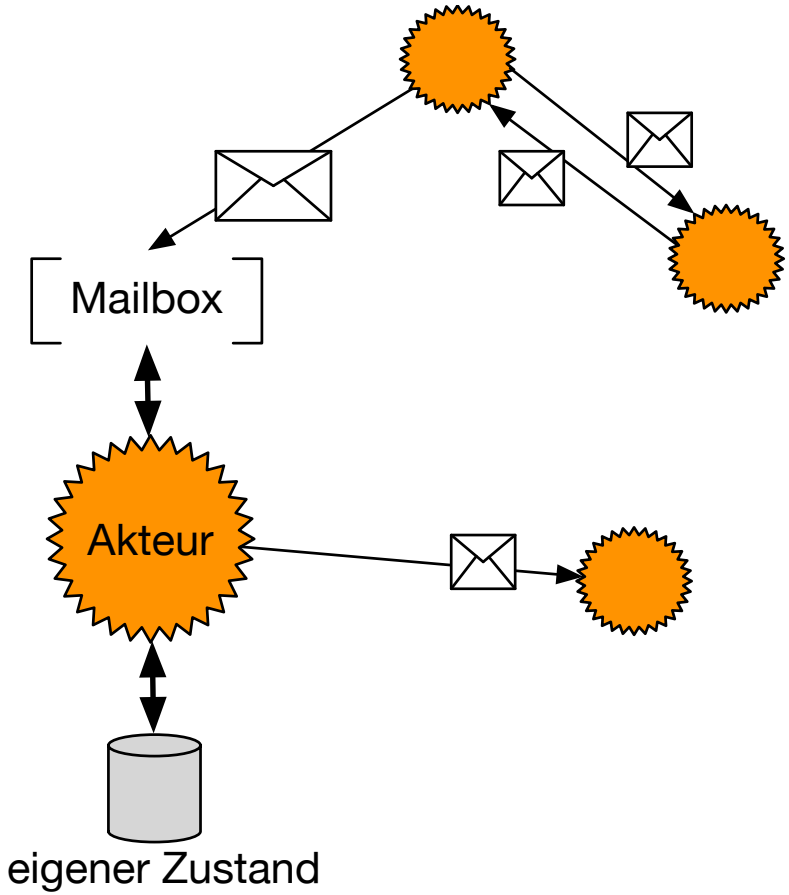
→ Open Telecom Platform (OTP)
→ Fault-Tolerant
→ „Let it crash“
→ Supervision-Trees



- die Fehlerverarbeitung ist interessant gelöst: sollte ein kritischer fehler aufreten, so startet man den prozess einfach neu.
- Dies ist durch sogennante provsion-trees möglich
- Ein Prozess kann mithilfe des OTP auf einfache art durch einen Supervisor beobachtet werden.
- Was im Fehlerfall geschehen soll, entscheidet einer von vielen Algorithmen und es sind damit beliebige schachtelungs-tiefen möglich. bspw. one-for-one

OTP / ACTOR-MODEL

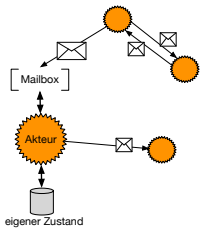
- → Concurrency-Model in Elixir
- → unabhängige Akteure
- → Message-Passing
- → FIFO-Verhalten von **Mailboxes**
- → Locks werden nicht gebraucht
- → Alternativen:
  - → Akka (Java/Scala)
  - → Akka.NET
  - → Pykka (Python)
  - → CAF (C++)
  - → Celluloid (Ruby)



└OTP / Actor-Model

OTP / ACTOR-MODEL

- → Concurrency-Model in Elixir
- → unabhängige Akteure
- → Message-Passing
- → FIFO-Verhalten von **Mailboxes**
- → Locks werden nicht gebraucht
- → Alternativen:
  - → Akka (Java/Scala)
  - → Akka.NET
  - → Pykka (Python)
  - → CAF (C++)
  - → Celluloid (Ruby)



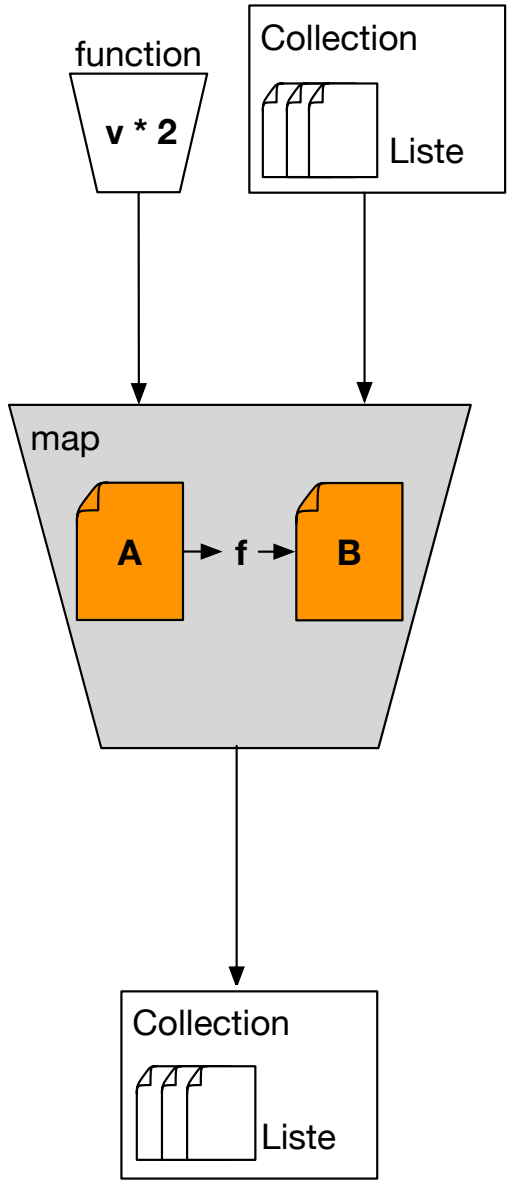
- akteuere können untereinander kommunizieren
- besitzen eigenen Zustand, locks werden damit überflüssig
- eigener Cache und Persistenz (DB) empfehlenswert
- deswegen klar getrennte Services
- Messages werden nach reihenfolge des eintreffens verarbeitet

LIST-PROCESSING IN ELIXIR: MAP (1)

→ Iteriert über Collection

→ Nutzen von transform-function

→ Ergebnis von gleichem oder verschiedenen Collection-Typ



LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`
→ **[2, 3, 4]**

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`
→ **[2, 3, 4]**
→ `iex> Enum.map [1, 2, 3], &(&1 * &1)`

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`
→ **[2, 3, 4]**
→ `iex> Enum.map [1, 2, 3], &(&1 * &1)`
→ **[1, 4, 9]**

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`
→ **[2, 3, 4]**
→ `iex> Enum.map [1, 2, 3], &(&1 * &1)`
→ **[1, 4, 9]**
→
```
iex> defmodule Math do
...> def multWithKey({k, v}), do: k * v
...> end
```
...

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

→ `iex> Enum.map [1, 2, 3], fn x -> x + 1 end`
→ **[2, 3, 4]**
→ `iex> Enum.map [1, 2, 3], &(&1 * &1)`
→ **[1, 4, 9]**
→
```
iex> defmodule Math do
...> def multWithKey({k, v}), do: k * v
...> end
```
...
→ `iex> list = Enum.with_index([1, 2, 3])`

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

```
→ iex> Enum.map [1, 2, 3], fn x -> x + 1 end
→ [2, 3, 4]
→ iex> Enum.map [1, 2, 3], &(&1 * &1)
→ [1, 4, 9]
→ iex> defmodule Math do
   ...> def multWithKey({k, v}), do: k * v
   ...> end
   ...
→ iex> list = Enum.with_index([1, 2, 3])
→ [{1, 0}, {2, 1}, {3, 2}]
```

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

```
→ iex> Enum.map [1, 2, 3], fn x -> x + 1 end
→ [2, 3, 4]
→ iex> Enum.map [1, 2, 3], &(&1 * &1)
→ [1, 4, 9]
→ iex> defmodule Math do
   ...> def multWithKey({k, v}), do: k * v
   ...> end
   ...
→ iex> list = Enum.with_index([1, 2, 3])
→ [{1, 0}, {2, 1}, {3, 2}]
→ iex> Enum.map list, &Math.multWithKey/1
```

LIST-PROCESSING IN ELIXIR: MAP (2)

iex = Interactive Elixir
& = function catch operator

```
→ iex> Enum.map [1, 2, 3], fn x -> x + 1 end
→ [2, 3, 4]
→ iex> Enum.map [1, 2, 3], &(&1 * &1)
→ [1, 4, 9]
→ iex> defmodule Math do
   ...> def multWithKey({k, v}), do: k * v
   ...> end
   ...
→ iex> list = Enum.with_index([1, 2, 3])
→ [{1, 0}, {2, 1}, {3, 2}]
→ iex> Enum.map list, &Math.multWithKey/1
→ [0, 2, 6]
```

ELIXIR-STREAMS (1)

→ große Datenstruktur kann lazy verarbeitet werden

→ Viele Funktionen aus `Enum`-Modul

→ `take`: Man erhält immer nur so viele Elemente wie benötigt

→ function-composition: Jeweiliges Element wird einmalig iteriert und dabei transformiert

→ siehe Clojure Reducers/Transducers

- nicht direkt mit parallelisierung zu tun, jedoch wichtige optimierung dabei
-

→ große Datenstruktur kann lazy verarbeitet werden
→ Viele Funktionen aus `Enum`-Modul
→ `take`: Man erhält immer nur so viele Elemente wie benötigt
→ function-composition: Jeweiliges Element wird einmalig iteriert und dabei transformiert
→ siehe Clojure Reducers/Transducers

ELIXIR-STREAMS (2)

|> = apply, forward pipe

```
→ iex> 1..10000
    |> Stream.map(&(&1 * &1))
    |> Stream.map(&(&1 + &1))
    |> Stream.map(&IO.inspect(&1))
    |> Enum.take(10)
```

ELIXIR-STREAMS (2)

|> = apply, forward pipe

→
```
iex> 1..10000
|> Stream.map(&(&1 * &1))
|> Stream.map(&(&1 + &1))
|> Stream.map(&IO.inspect(&1))
|> Enum.take(10)
```

→ 2
8
18
..
200
[2, 8, 18, 32, 50, 72, 98, 128, 162, 200]

ELIXIR-PROZESSE ERZEUGEN

```
defmodule Parallel do
 def pmap(collection, fun) do
   me = self
   collection
   |> Enum.map(fn (elem) ->
       spawn fn -> (send me,  self, fun.(elem) ) end
      end)
   |> Enum.map(fn (pid) ->
       receive do  ^pid, result  -> result end
      end)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

ELIXIR-PROZESSE ERZEUGEN

```
defmodule Parallel do
 def pmap(collection, fun) do
   me = self
   collection
   |> Enum.map(fn (elem) ->
       spawn fn -> (send me,  self, fun.(elem) ) end
      end)
   |> Enum.map(fn (pid) ->
       receive do  ^pid, result  -> result end
      end)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

→ kann ebenso fehleranfällig sein

→ kein Supervisioning, manuelles Error-Handling, Timeouts?

OTP-ABSTRACTION: TASKS (1)

→ Task stellt einen simplen Background-Process dar

→ OTP ist sehr mächtig, jedoch nicht immer eigene Implementierung benötigt

→ Ein Elixir Process ist häufiger Anwendungsfall

→ ist in Elixir-Core-Package vorhanden

OTP-ABSTRACTION: TASKS (2)

```
defmodule Parallel do
  def pmap(collection, func) do
    collection
    |> Enum.map(&(Task.async(fn -> func.(&1))))
    |> Enum.map(&Task.await/1)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

OTP-ABSTRACTION: TASKS (2)

```
defmodule Parallel do
  def pmap(collection, func) do
    collection
    |> Enum.map(&(Task.async(fn -> func.(&1))))
    |> Enum.map(&Task.await/1)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

→ noch cleaner als mit `spawn`

→ möglich Fallstricke sind bereits optimiert

→ weil otp direkt für supervisioning nutzbar

FAZIT

FAZIT: FUNCTIONAL PROGRAMMING

+ +

→ immutable State eignet sich perfekt für parallel

→ Higher-Order-Programming: composeability of behaviors

→ sehr deklarativ

- -

→ Je nachdem steile Lernkurve

→ Man muss viele bisher eingesetzte Methodiken komplett überdenken

└Fazit: Functional Programming

+ +
- → immutable State eignet sich perfekt für parallel
- → Higher-Order-Programming: composeability of behaviors
- → sehr deklarativ

- -
- → Je nachdem steile Lernkurve
- → Man muss viele bisher eingesetzte Methodiken komplett überdenken

- Je nach Sprache, eher erinnert Programmierung eher an zusammenstellung von Verhalten
- HOP, beschäftigt sich genau mit diesem Thema
- Art und Abfolge der Anweisung kann genau und deklarativ bestimmt werden
- Bei Nutzen des OTP nebenläufiges und verteiletes System von hause aus
- Häufige Bugs bei Threads-Programming (Deadlocks, Race-Conditions,..) werden einfach umgangen
- Wenige Sprachen helfen Concurrency wirklich leichter zu machen
- meist keine direkte Unterstützung für reine Parallelisierung