Elixir

Sebastian Wagner, HAW Hamburg 19.10.2016

Wie dieser Vortrag funktioniert

Gliederung

- Motivation und Einführung
- Sprachkonzepte
- Praxis
- Fazit

Motivation

Bildquelle: Erlang: The Movie, Telefonaktiebolaget L M Ericsson, Kista, 1990

- Kompiliert in Erlang Bytecode
  - Läuft auf Erlang VM (BEAM)
- Dynamisch getypt
- Message Passing
- Unterstützt Nebenläufigkeit
- Homoikonisch

Erlang

- Erlang wurde 1986 als proprietäre Sprache bei Ericsson zunächst von Joe Armstrong entwickelt
- Ziel: Telefonanlagen programmieren
  - Weiche Embedded-Echtzeitsysteme
  - Hohe Verfügbarkeit
  - Aktualisierungen ohne Downtime
- Damals eine ziemliche Nische

vgl. Armstrong, Joe. "A history of Erlang." Proceedings of the third ACM SIGPLAN conference on History of programming languages. ACM, 2007.

„Erlang was designed for writing concurrent programs that “run forever.” Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system.„

– Joe Armstrong

Armstrong 2007 S.1

Was ist seit 1986 passiert?

- Das Internet wie wir es heute kennen
- Herb Sutter: „The Free Lunch is over“[1]
- Erfahrungen mit Programmierkonzepten[2]
- OOP und seine Konsequenzen
- Metaprogrammierung in Sprachen wie Ruby
- Neue, verteilte Architekturmuster
- Ansteigende Popularität funktionaler Konzepte

1) Sutter, Herb. The Free Lunch Is Over. *http://www.gotw.ca/publications/concurrency-ddj.htm* 2005, Abruf 3.10.2016
2) vgl. Thomas, Dave. *Programming Elixir: Functional |> Concurrent |> Pragmatic |> Fun*. Pragmatic Bookshelf, 2014. S. XIII ff.

„Our CPUs are not getting any faster. Instead, our computers get more and more cores. This means new software needs to use as many cores as it can if it is to maximize its use of the machine. This conflicts directly with how we currently write software.“

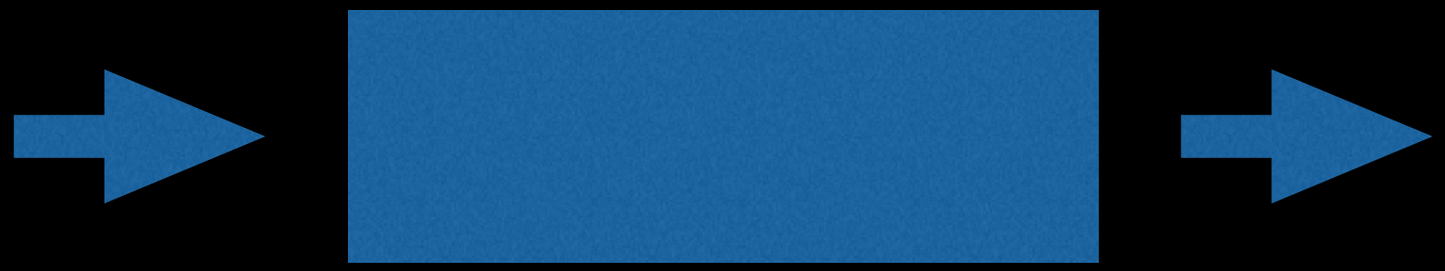
–José Valim

Thomas 2014 S. XI

Sprachkonzepte

Prozesse

```
# cat heuhaufen | grep nadel
```



Eingabe Transformation Ausgabe

[1,2,3,4] [2,4,6,8]

[3,4,2,1] [1,2,3,4]

"https://google.de" <html> …

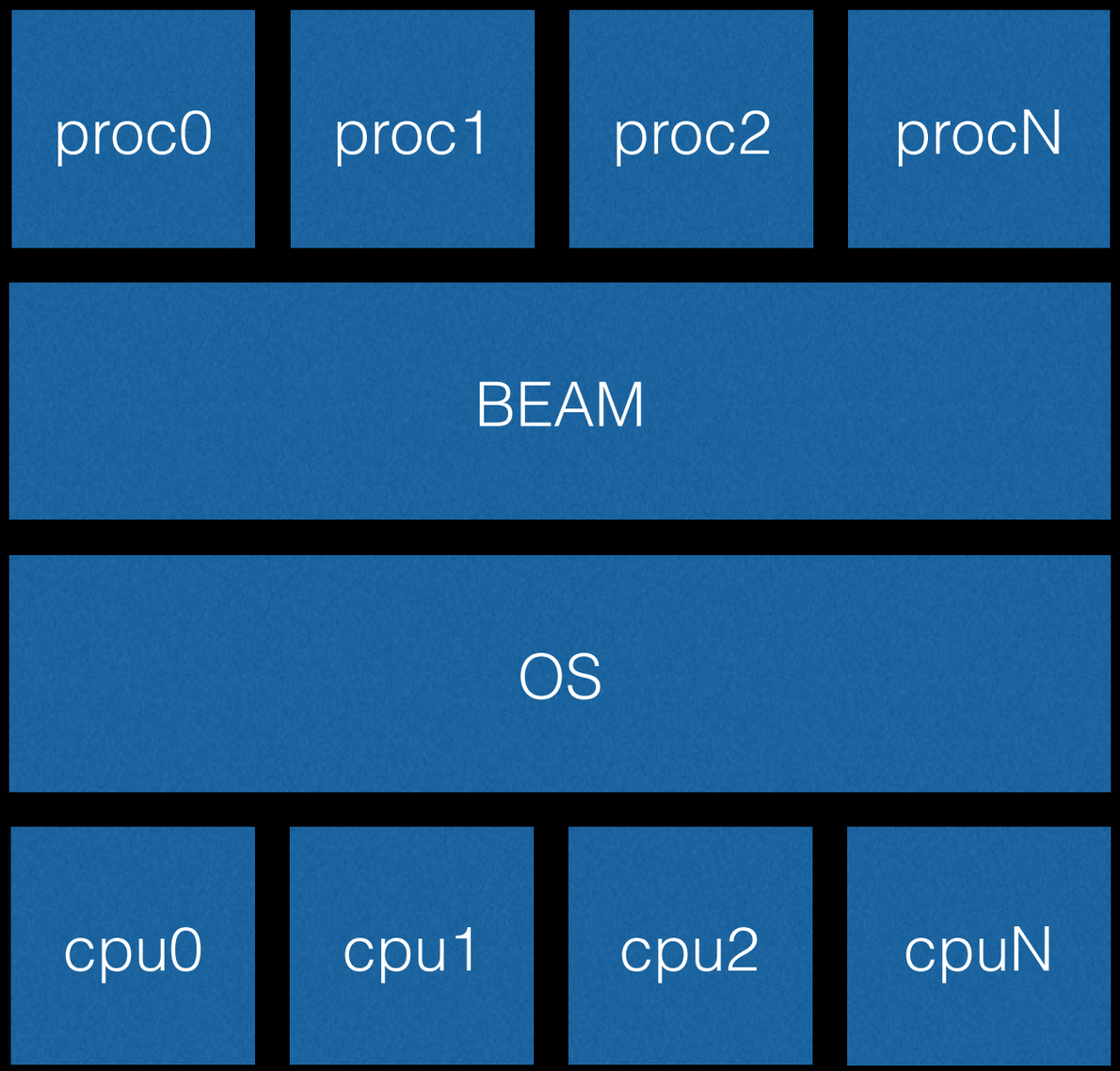
„Erlang was designed for writing concurrent programs that “run forever.” Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system.„

– Joe Armstrong

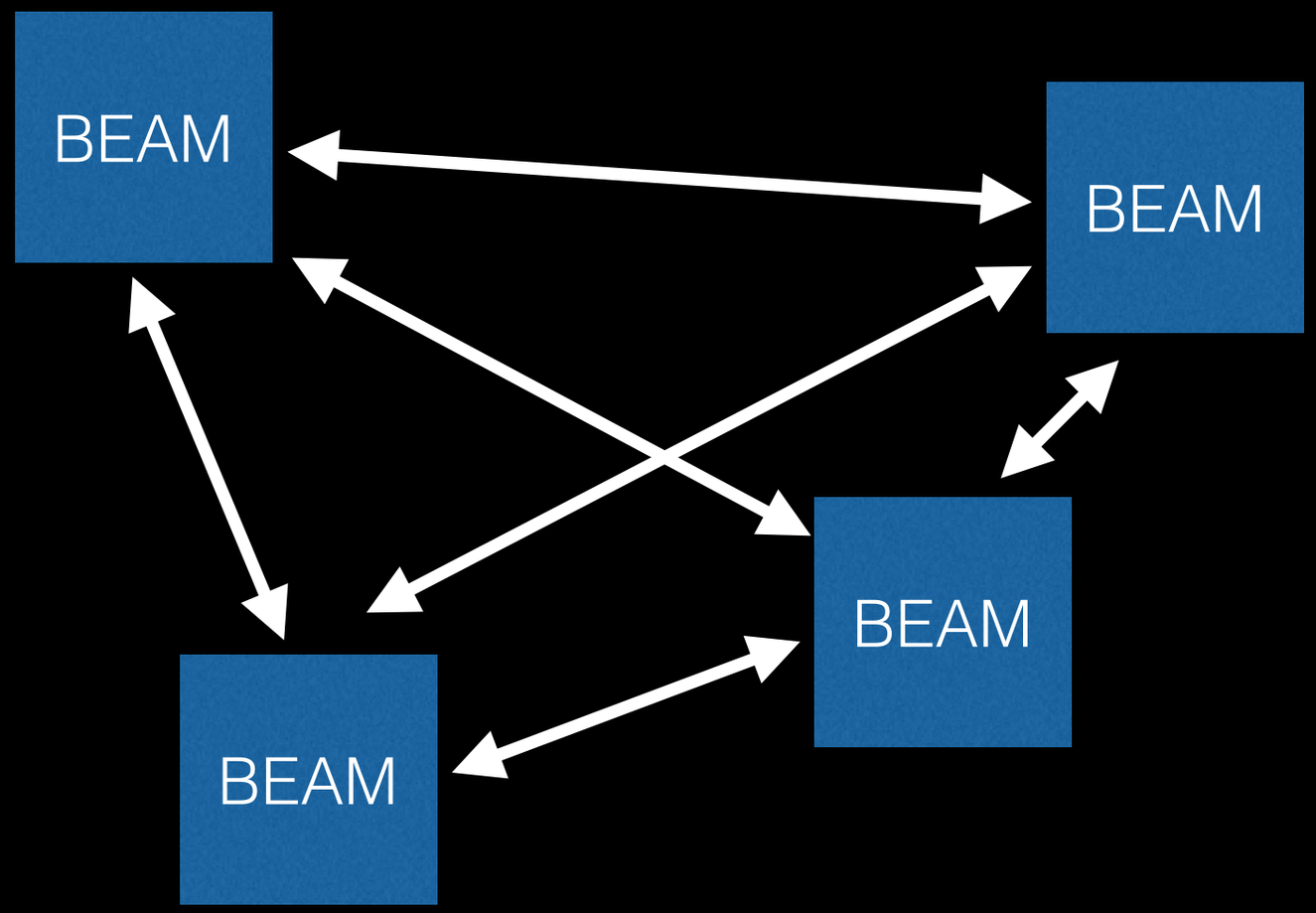
Armstrong 2007 S.1

Implikationen

- Kein geteilter Speicher (shared nothing)
  - Viele typische Probleme stellen sich gar nicht erst
- Viele billige, kleine Prozesse
  - Probleme sind gekapselt
  - Kann leicht verteilt werden



vgl. Erlang Efficiency Guide http://erlang.org/doc/efficiency_guide/processes.html Abruf 5.10.2016



vgl. Erlang Efficiency Guide

OTP

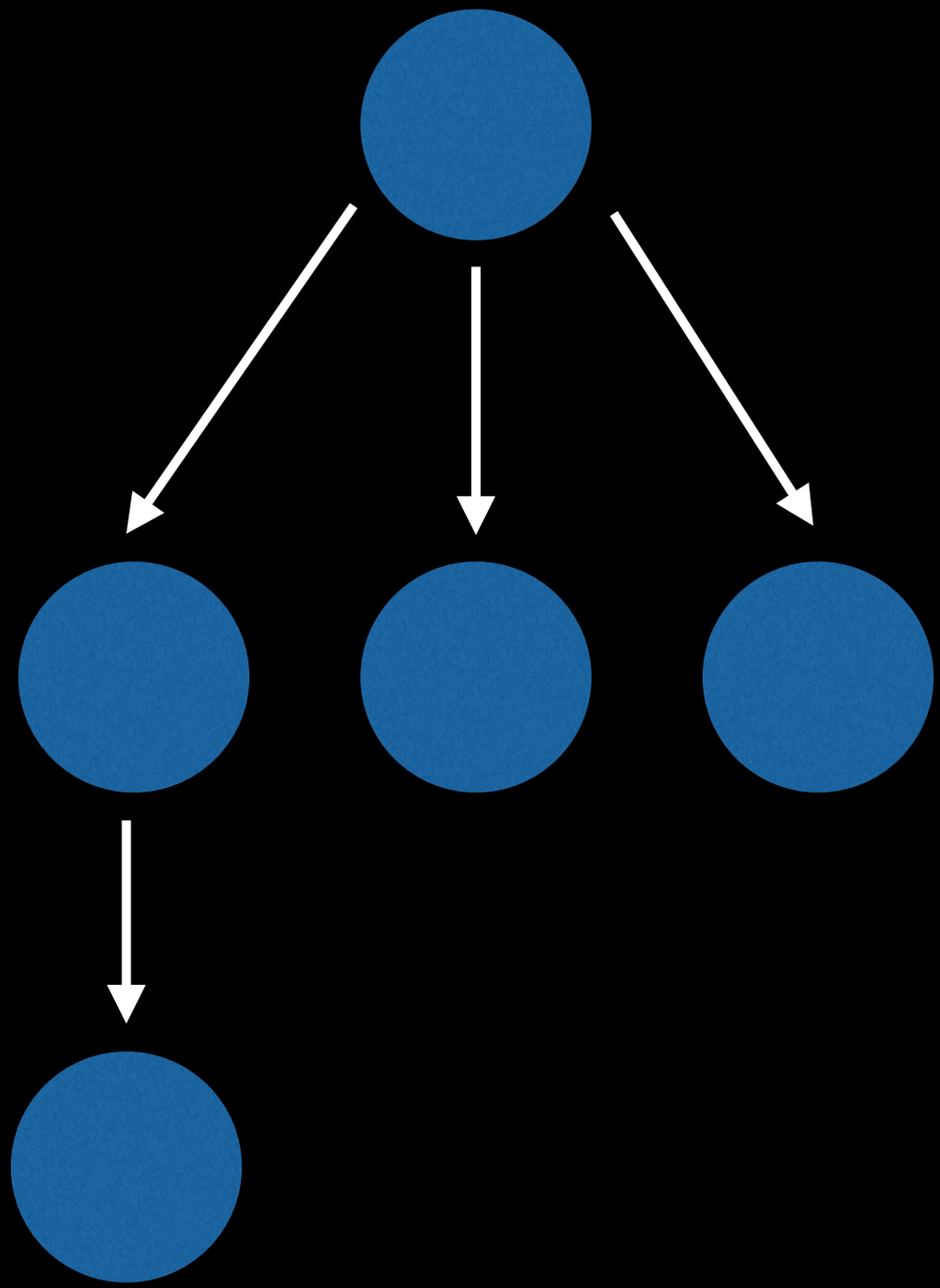
Was fehlt noch?

Programs that run forever.

OTP

- Open Telecom Platform
- Baumstruktur verbundener Prozesse
- Überwachung durch Supervisor-Prozesse
- Ggf. Automatischer Neustart

vgl. Hébert, Fred. Learn You Some Erlang for Great Good!: A Beginner's Guide. No Starch Press, 2013. Kapitel „What is OTP?“



Konsequenzen

- Ein Crash beschränkt sich auf einen Prozess
  - Rest der Anwendung bleibt laufen
- Crashes sind etwas gutes
- Konzentration auf den „Happy Path“
  - Besserer Fokus auf das eigentliche Problem

Code

Wichtige Datentypen

```
boolean = true
integer = 1
float   = 1.5
atom    = :wuppifluppi
tuple   = {:ok, 3}
list    = [1,2,3]
string  = „abc“ # auch binary!
map     = %{a: 3, b: 4}
```

Immutability

- Alles ist immutable - vereinfacht Denken enorm
- … Namen können aber neu gebunden werden
- Ggf. werden Werte effizient kopiert
  - Garbage Collection auf Prozessebene
    - Prozess hat eigenen Heap

```
iex(1)> liste = [1, 2, [3]]
[1, 2, [3]]
iex(2)> neue_liste = List.flatten(liste)
[1, 2, 3]
iex(3)> liste
[1, 2, [3]]
```


vgl. Thomas 2014 S. 19-22

Funktionen

```elixir
def add(a, b) do
    a + b
end
```

Elixir

```ruby
def add(a, b)
    a + b
end
```

Ruby

```python
def add(a, b):
    return a + b
```

Python

Funktionen

```
@doc """
Allows you to get parsed JSON from a URL. JSON is returned as a map.
"""
def get(url) do
  url
  |> do_get_request
  |> check_status_code
  |> decompress_response
  |> parse_json
end
```

Pattern Matching

```
def check_status_code(response) do
    # Only accept 200
    200 = response.status_code
    response
end
```

Guards

```elixir
def say(power_level) when power_level > 9000, do: IO.puts("It's over 9000!")
def say(power_level), do: IO.puts("Meh.")
```

- Metaprogrammierung erlaubt direkten Zugriff auf den Syntaxbaum[1]
  - Ermöglicht mächtige Makros und DSLs
    - Viele Sprachfeatures sind selbst als Macros implementiert[2]
- Protocols ermöglichen Polymorphie[3]

1) vgl. http://elixir-lang.org/getting-started/meta/quote-and-unquote.html Abruf 5.10.2016
2) vgl. http://elixir-lang.org/getting-started/meta/macros.html Abruf 5.10.2016
3) vgl. http://elixir-lang.org/getting-started/protocols.html Abruf 5.10.2016

Konsequenzen

- Viele traditionelle Kontrollstrukturen entfallen
- Conditionals werden zu Pattern-Matching
- Schleifen werden durch funktionale Konzepte ersetzt
- Exceptions werden selten behandelt
  - Es wird so schnell gecrasht wie möglich
- Kompakter, lesbarer, einfach testbarer Code

Praxis

Skalierbarkeit

- Ca. 40% der 3G/LTE Infrastruktur läuft mit Erlang-Software[1]

- WhatsApp (2012)
  - Kleines Team
  - 2M Verbindungen auf einem Server[2]

1) Inside Erlang – creator Joe Armstrong tells his story, https://www.youtube.com/watch?v=2T3dgKhoU6U Abruf 6.10.2016
2) 1 million is so 2011 https://blog.whatsapp.com/196/1-million-is-so-2011 Abruf 6.10.2016

Skalierbarkeit

- Phoenix
  - Web Framework in Elixir
  - Basiert auf Erfahrungen mit vorigen Frameworks wie Rails
  - Fokus auf Performance und Echtzeit-Kommunikation
  - 2M parallele Websocket-Verbindungen[1] auf einzelnem Server möglich
- Weitere Skalierung über Distributed Erlang über niederlatentes Netzwerk

1) The Road to 2 Million Websocket Connections in Phoenix, http://www.phoenixframework.org/blog/the-road-to-2-million-websocket-connections Abruf 6.10.2016

Tooling

- Developer Tooling: Mix, Hex, ExUnit, ExDoc, diverse Analysetools wie Credo
- Gute Editor-Integration mit autocompletion und Dokumentation dank alchemist in z.B. emacs und Atom
- Type-Checking über Dialyxir
- Performance-Analyse über z.B. Flamegraphs mit eflame
- NIFs erlauben es nativen Code einzubinden
  - Können aber gesamte VM crashen!
- Releases über z.B. Distillery möglich
  - Kann nahtlos mit Docker deployed werden

Einsatzgebiete

- Webanwendungen
  - Standard CRUD Anwendungen / REST APIs
  - Real-Time Web (wird immer wichtiger)
- Backend-Services und APIs
- Gameserver
- Telekommunikation
  - Embedded/IoT - Nerves
  - Netzwerk-Backbone

Abgrenzung von ähnlichen Sprachen

- Streng genommen keine rein funktionale Sprache wie z.B. Haskell
  - Kein so mächtiges Typensystem, dafür zugänglich
  - Keine bloße Event-Loop wie z.B. NodeJS, eigener Scheduler
  - Kein „Impedance Mismatch“ wie bei einigen JVM-Sprachen, BEAM wurde für diesen Zweck gebaut
  - Kann auf OTP zurückgreifen
  - Modernes tooling, aktive Community

Fazit

Fazit

- Zugängliche Sprache für fortgeschrittene Konzepte
- Gut geeignet für hochverfügbare, ggf. verteilte Systeme
    - Ideal bei Event-Streams, Message Queues, Soft-Realtime Problemen u.ä.
    - Aktives, modernes Ökosystem
    - Beste Aussichten nach heutigem Stand
- Viele neue Ideen auf solidem Fundament, hohe Produktivität
- Nicht so gut geeignet für z.B. low-level Programmierung, bloßes Number-Crunching (NIFs), Native GUIs etc.

Weiterführendes

- Literatur
  - Dave Thomas, Programming Elixir
  - Saša Jurić, Elixir in Action
  - Chris McCord et al., Programming Phoenix
  - Chris McCord, Metaprogramming Elixir
- Meetups
  - hh.ex Meetup, jeden letzten Dienstag im Monat
- Konferenzen (ElixirConf, ElixirDaze, Erlang Factory, …)
- Libraries und Tooling: Awesome Elixir, hex.pm