

# FUNAR Technologien



# Haskell

- statisches Typsystem, Typklassen
- effektvolle Berechnungen über explizite Monaden
- viele Erweiterungen
- nicht-strikte Auswertung
- viele Optionen für Parallelisierung und Nebenläufigkeit
- eigene Runtime
- hochoptimierender Compiler (Native Code)
- weltweite Community entwickelt Compiler



- statisches Typsystem
- mächtiges Modulsystem
- imperative Features
- eigene Runtime  
(Bytecode, Native Code)
- optimierender Compiler
- Entwicklung koordiniert von  
INRIA, OCamlLabs

# Beispiel

[https://github.com/janestreet/async\\_smtp](https://github.com/janestreet/async_smtp)

```
module Spoolable = struct
  module type S = sig
    (** [Spoolable.Metadata.t] should be smallish since it is read and written more
        frequently than [Spoolable.Data.t]. *)
    module Metadata : sig
      type t

      (** [of_string] and [to_string] are used to persist and read [t] on disk. *)
      include Stringable.S with type t := t
    end

    (** [Spoolable.Data.t] is where the "real" data lives and it allows for data-specific
        [load] and [save] functionality. *)
    module Data : sig
      type t

      val load : string -> t Deferred.Or_error.t
      val save : ?temp_file:string -> t -> string -> unit Deferred.Or_error.t
    end

    (** [Queue.t] is an enumerable type that represents the available queues and the
        mapping to directory names on-disk. *)
    module Queue : sig
      type t [@@deriving sexp, enumerate, compare]

      val to_dirname : t -> string
    end

    module Name_generator : Name_generator.S

    (** All operations that touch disk are passed through [Throttle.enqueue] *)
    module Throttle : sig
      val enqueue : (unit -> 'a Deferred.t) -> 'a Deferred.t
    end
  end
end
```

# Beispiel

```
module Make_base (S : Multispool_intf.Spoolable.S) = struct
  include Shared

  type t = spool [@@deriving sexp]

  let dir t = t

  let load_metadata path =
    S.Throttle.enqueue (fun () ->
      Deferred.Or_error.try_with (fun () ->
        let%bind contents = Reader.file_contents path in
        return (S.Metadata.of_string contents)))

  let save_metadata ?temp_file ~contents path =
    S.Throttle.enqueue (fun () ->
      Deferred.Or_error.try_with (fun () ->
        Writer.save path ?temp_file ~contents ~fsync:true))

  module Data_file = struct
    type t =
      { spool : spool
      ; name : string
      }

    let create spool name = { spool; name }
    let path t = data_dir_of t.spool ^/ t.name
    let load t = S.Throttle.enqueue (fun () -> S.Data.load (path t))
    let save t ~contents = S.Throttle.enqueue (fun () -> S.Data.save contents (path t))

    let stat t =
      Deferred.Or_error.try_with (fun () ->
        S.Throttle.enqueue (fun () -> Unix.stat (data_dir_of t.spool ^/ t.name)))
  end
end
```



- dynamisch getypt
- Actor-Modell
- inhärent verteilt
- spezialisiert auf fehlertolerante Systeme
- umfangreiches Ökosystem für Deployment und Betrieb
- eigene Runtime (Bytecode, auch Native Code)
- Entwicklung koordiniert von Ericsson

# Beispiel

<https://github.com/zotonic/zotonic>

```
-module(mod_tkvstore).  
-author("Marc Worrell <marc@worrell.nl>").  
-behaviour(gen_server).  
  
-include_lib("zotonic_core/include/zotonic.hrl").  
  
%% @doc Fetch the persistent data of a type/key  
pid_observe_tkvstore_get(Pid, #tkvstore_get{} = Message, _Context) ->  
    gen_server:call(Pid, Message).  
  
%% @doc Fetch persistent data, first check the data dict that is still being written  
handle_call(#tkvstore_get{type=Type, key=Key}, _From, State) ->  
    case dict:find({Type, Key}, State#state.data) of  
        {ok, Data} ->  
            % Data is being written, return the data that is not yet in the store  
            {reply, Data, State};  
        error ->  
            {reply, m_tkvstore:get(Type, Key, State#state.context), State}  
    end;
```



elixir

- alternative Sprache für die Erlang-Plattform
- inspiriert durch Ruby
- interoperabel mit Erlang
- eigenes Build-/Package-Tooling



# Beispiel

<https://github.com/elixir-ecto/ecto>

```
defmodule Ecto.Repo.Registry do

  use GenServer

  def associate(pid, value) when is_pid(pid) do
    GenServer.call(__MODULE__, {:associate, pid, value})
  end

  @impl true
  def handle_call({:associate, pid, value}, _from, table) do
    ref = Process.monitor(pid)
    true = :ets.insert(table, {pid, ref, value})
    {:reply, :ok, table}
  end

  @impl true
  def handle_info({:DOWN, ref, _type, pid, _reason}, table) do
    [{^pid, ^ref, _}] = :ets.lookup(table, pid)
    :ets.delete(table, pid)
    {:noreply, table}
  end
end
```



# Swift

- hybride OO-/FP-Sprache
- statisches Typsystem
- entwickelt von Apple
- eng mit Objective-C / Cocoa integriert
- eigene Runtime / Reference Counting
- Native Code

# Beispiel

```
indirect enum Diagram {
    case primitive(CGSize, Primitive)
    case beside(Diagram, Diagram)
    case below(Diagram, Diagram)
    case attributed(Attribute, Diagram)
    case align(CGPoint, Diagram)
}

extension Diagram {
    var size: CGSize {
        switch self {
            case .primitive(let size, _): return size
            case .attributed(_, let x): return x.size
            case let .beside(l, r):
                let sizeL = l.size
                let sizeR = r.size
                return CGSize(width: sizeL.width + sizeR.width,
                               height: max(sizeL.height, sizeR.height))
            case let .below(l, r):
                return CGSize(width: max(l.size.width, r.size.width),
                               height: l.size.height + r.size.height)
            case .align(_, let r): return r.size
        }
    }
}
```

# Kotlin

- primär OO-Sprache
- Funktionstypen  
(keine “Higher-Kinded Types”)
- statisches Typsystem
- entwickelt von JetBrains
- JVM



# Beispiel

```
fun <A> Stream<A>.exists(p: (A) -> Boolean): Boolean =  
    when (this) {  
        is Cons -> p(this.h()) || this.t().exists(p)  
        else -> false  
    }
```

```
fun <A, B> Stream<A>.foldRight(  
    z: () -> B,  
    f: (A, () -> B) -> B  
) : B = // <1>  
    when (this) {  
        is Cons -> f(this.h()) { t().foldRight(z, f) } // <2>  
        else -> z()  
    }
```

```
fun <A> Stream<A>.exists2(p: (A) -> Boolean): Boolean =  
    foldRight({ false }, { a, b -> p(a) || b() })
```



# Racket

- Baukasten für Sprachen
- enthält mehrere spezielle Lehrsprachen
- Abkömmling von Scheme/Lisp
- ungetypte und getypte Varianten
- eigene Runtime  
(Byte-Code, Native Code, JIT)

# Beispiel

<https://racket-lang.org/>

## #lang racket

```
(require 2htdp/image) ; draw a picture
(let sierpinski ([n 8])
  (cond
    [(zero? n) (triangle 2 'solid 'red)]
    [else (define t (sierpinski (- n 1)))
```

## #lang typed/racket

```
;; Using higher-order occurrence typing
(define-type SrN (U String Number))
(: tog ((Listof SrN) -> String))
(define (tog l)
  (apply string-append (filter string? l)))
```

## #lang racket/gui

```
(define f (new frame% [label "Guess"]))
(define n (random 5)) (send f show #t)
(define ((check i) btn evt)
  (message-box "." (if (= i n) "Yes" "No")))
(for ([i (in-range 5)])
```

## #lang scribble/base

```
@; Generate a PDF or HTML document
@title{Bottles: @italic{Abridged}}
@ (apply
  itemlist
  (for/list ([n (in-range 100 0 -1)])
```

## #lang datalog

```
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :-
  parent(A, C), ancestor(C, B).
parent(john, douglas).
parent(bob, john).
```

## #lang web-server/insta

```
;; A "hello world" web server
(define (start request)
  (response/xexpr
    '(html
      (head (title "Racket"))
```



F#

- .NET-Abkömmling von OCaml
- allerdings ohne dessen Modulsystem
- statisches Typsystem
- ... mit Maßeinheiten
- integriert in Visual Studio
- *type providers*
- Entwicklung bei Microsoft koordiniert



# Beispiel

<http://fsharp.github.io/FSharp.Data/>

```
#r "FSharp.Data.dll"
open FSharp.Data

let wb = WorldBankData.GetDataContext()
let uk = wb.Countries.`United Kingdom`
uk.Indicators|
```

- ⌘ Average grace period on new external debt commitments, official (years)
- ⌘ Average grace period on new external debt commitments, private (years)
- ⌘ Average grant element on new external debt commitments (%)
- ⌘ Average grant element on new external debt commitments, official (%)
- ⌘ Average grant element on new external debt commitments, private (%)
- ⌘ Average interest on new external debt commitments (%)
- ⌘ Average interest on new external debt commitments, official (%)
- ⌘ Average interest on new external debt commitments, private (%)
- ⌘ Average maturity on new external debt commitments (years)



- hybride OO-/FP-Sprache
- statisches Typsystem
- Java-Plattform
- Entwicklung koordiniert bei  
Lightbend (Scala 2)  
EPFL (Scala 3)
- Fokus auf Weiterentwicklung

# Beispiel

```
package de.active.funar

import cats.effect.{ExitCode, IO, IOApp, Resource}
import cats.implicits._
import fs2.{io, text, Stream}
import java.nio.file.Paths
import java.util.concurrent.Executors
import scala.concurrent.ExecutionContext

object Converter extends IOApp {
  private val blockingExecutionContext =
    Resource.make(IO(ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(2))))(ec => IO(ec.shutdown()))

  val converter: Stream[IO, Unit] = Stream.resource(blockingExecutionContext).flatMap { blockingEC =>
    def fahrenheitToCelsius(f: Double): Double =
      (f - 32.0) * (5.0/9.0)

    io.file.readAll[IO](Paths.get("testdata/fahrenheit.txt"), blockingEC, 4096)
      .through(text.utf8Decode)
      .through(text.lines)
      .filter(s => !s.trim.isEmpty && !s.startsWith("//"))
      .map(line => fahrenheitToCelsius(line.toDouble).toString)
      .intersperse("\n")
      .through(text.utf8Encode)
      .through(io.file.writeAll(Paths.get("testdata/celsius.txt"), blockingEC))
  }

  def run(args: List[String]): IO[ExitCode] =
    converter.compile.drain.as(ExitCode.Success)
}
```



# Clojure

- modernes Lisp
- dynamisch getypt
- Java-Plattform
- Entwicklung koordiniert bei Cognitect
- Fokus auf Stabilität

# Beispiel

<https://github.com/active-group/vfei>

```
; value is a map
(define-record-type ListFormat
  (^{:doc "Make a list format for a VFEI data item."}
    make-list-format size)
  list-format?
  [size list-format-size])

(define-record-type ArrayFormat
  (^{:doc "Make an array format for a VFEI data item."}
    make-array-format element-format size)
  array-format?
  [element-format array-format-element-format
   size array-format-size])

(define-record-type DataItem
  (^{:doc "Make a VFEI data item."}
    really-make-data-item name format value)
  data-item?
  [name data-item-name
   format data-item-format
   value data-item-value])

(defn make-data-item
  "Assert that list length matches format."
  [name format value]
  (cond
    (list-format? format)
    (when (not= (list-format-size format) (count value))
      (c/error `make-data-item "list length does not match number of parsed values" name format (count value) value))

    (array-format? format)
    (when (not= (array-format-size format) (count value))
      (c/error `make-data-item "array size does not match number of parsed values" name format (count value) value)))

  (really-make-data-item name format value))
```

# Nach Plattform

| Plattform                   | Sprache |
|-----------------------------|---------|
| Java                        | Clojure |
| Java                        | Scala   |
| .NET                        | F#      |
| Apple, Linux                | Swift   |
| Erlang (Unix, Windows)      | Elixir  |
| Erlang                      | Erlang  |
| Haskell (Unix, Windows)     | Haskell |
| OCaml (Unix)                | OCaml   |
| Racket (Unix, Windows, Mac) | Racket  |

# Statisch/dynamisch getypt

| Typsystem                 | Sprache |
|---------------------------|---------|
| dynamisch                 | Clojure |
| dynamisch                 | Elixir  |
| dynamisch                 | Erlang  |
| dynamisch (auch statisch) | Racket  |
| statisch                  | F#      |
| statisch, Typklassen      | Haskell |
| statisch, Module          | OCaml   |
| statisch, OO-Integration  | Scala   |
| statisch, OO-Integration  | Swift   |
| statisch, OO-Integration  | Kotlin  |

# Merkmal pro Sprache

| Merkmal                            | Sprache         |
|------------------------------------|-----------------|
| Lisp                               | Clojure, Racket |
| Erlang, andere Syntax              | Elixir          |
| verteilte, fehlertolerante Systeme | Erlang          |
| Sprache der Wahl auf .NET          | F#              |
| „fancy types“                      | Haskell         |
| Modulsystem                        | OCaml           |
| Sprach-Baukasten                   | Racket          |
| hybrid OO/FP auf Java-Plattform    | Scala           |
| hybrid OO/FP auf Apple-Plattform   | Swift           |



# Grund pro Sprache

| Grund                                   |                        |
|---|------------------------|
| Performance, große Projekte             | OCaml                  |
| Performance, Parallelismus, fancy types | Haskell                |
| Verteilung, Fehlertoleranz              | Erlang, Elixir         |
| .NET                                    | F#                     |
| Java                                    | Clojure, Scala, Kotlin |
| DSL                                     | Racket                 |
| Apple                                   | Swift                  |
| Android                                 | Kotlin                 |

# Nicht-Strikte Auswertung

```
data Tree a = Node a [Tree a]
```

```
maptree f (Node a sub) =  
    Node (f a) (map (maptree f) sub)
```

```
reptree :: (a -> [a]) -> a -> Tree a
```

```
reptree f a =  
    Node a (map (reptree f) (f a))
```

```
gametree :: Board -> Tree Board
```

```
gametree p = reptree moves p
```

# Endrekursion / Tail Calls







```
reverse l = rev l []
```

**where**

```
rev [] a = a
```

```
rev (x:xs) a = rev xs (x:a)
```

# Tail Calls vs. Plattform

| Plattform                         | “Proper Tail Calls”   |
|-----------------------------------|---|
| JavaScript                        | Standard:  ,<br>Implementierungen:  |
| Java (Scala, Clojure)             |   |
| .NET (F#)                         |   |
| Racket, OCaml, Haskell,<br>Erlang |   |
| Swift                             |    |

# Eingebettete DSLs

| Sprache | DSL-Mechanismus            |
|---------|----------------------------|
| Racket  | Makros                     |
| Clojure | Makros                     |
| Elixir  | Makros                     |
| Haskell | Monaden + Typklassen       |
| Scala   | Monaden + Implicits        |
| Kotlin  | Monaden + Receiver Objects |

# Varianten

| Sprache     | Varianten                                 |
|-------------|---|
| Haskell     | Frege (JVM)<br>ghc -> JavaScript (ab 9.6) |
| Haskell/Web | Purescript, Elm                           |
| OCaml       | Reason (OCaml, JS),<br>Js_of_ocaml (JS)   |
| Scala       | ScalaJS (JS)                              |
| Clojure     | ClojureScript (JS)                        |
| F#          | Fable (JS)                                |
| Kotlin      | Kotlin for JavaScript                     |