Von Java zu Haskell

Michael Sperber
@sperbsen

- Individualsoftware
- branchenunabhängig
- Scala, Clojure, Erlang, Haskell, OCaml, F#
- Schulungen, Coaching

www.active-group.de

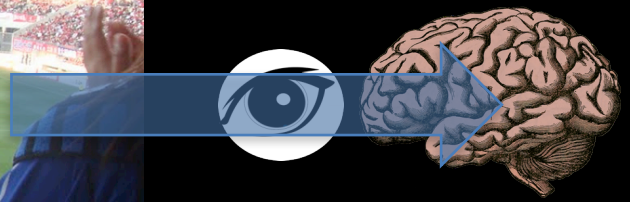
funktionale-programmierung.de

@ctive group

http://bobkonf.de/2017/

Eine Welt von Objekten

Realität und Schnappschüsse



Haskell

```
data Animal =
    Dillo { alive :: Bool, weight :: Int }
  | Rattlesnake { thickness :: Int, len :: Int }
  | Mouse { blood :: Int, size :: Int }
  deriving Show

h1 = [ Dillo { alive = True, weight = 10 },
       Rattlesnake { thickness = 5, len = 100 },
       Mouse { blood = 5, size = 17 },
       Rattlesnake { thickness = 7,
                     len = 200 } ]
```

Haskell

```
runOver :: Animal -> Animal
runOver (d@Dillo{})       =
  Dillo { alive = False, weight = weight d }
runOver (r@Rattlesnake{}) = r { thickness = 0 }
runOver (m@Mouse{})       = m { blood = 0 }

isDeadRattlesnake :: Animal -> Bool
isDeadRattlesnake (Dillo{}) = False
isDeadRattlesnake
  (Rattlesnake { thickness = t }) = t == 0
isDeadRattlesnake (Mouse{}) = False
```

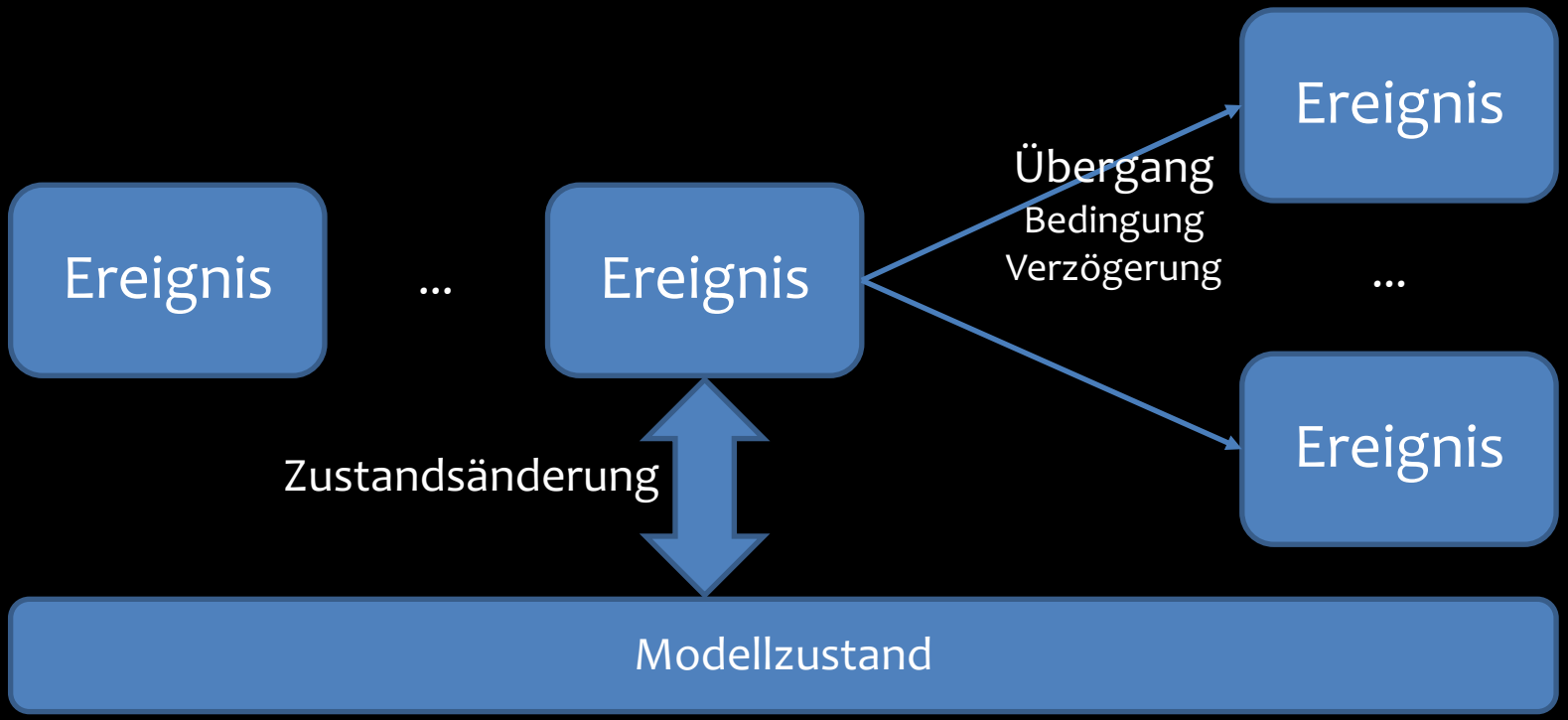
Haskell

```
runOverAll :: [Animal] -> [Animal]
runOverAll a = map runOver a

deadRattlesnakes :: [Animal] -> [Animal]
deadRattlesnakes a =
  filter isDeadRattlesnake a
```

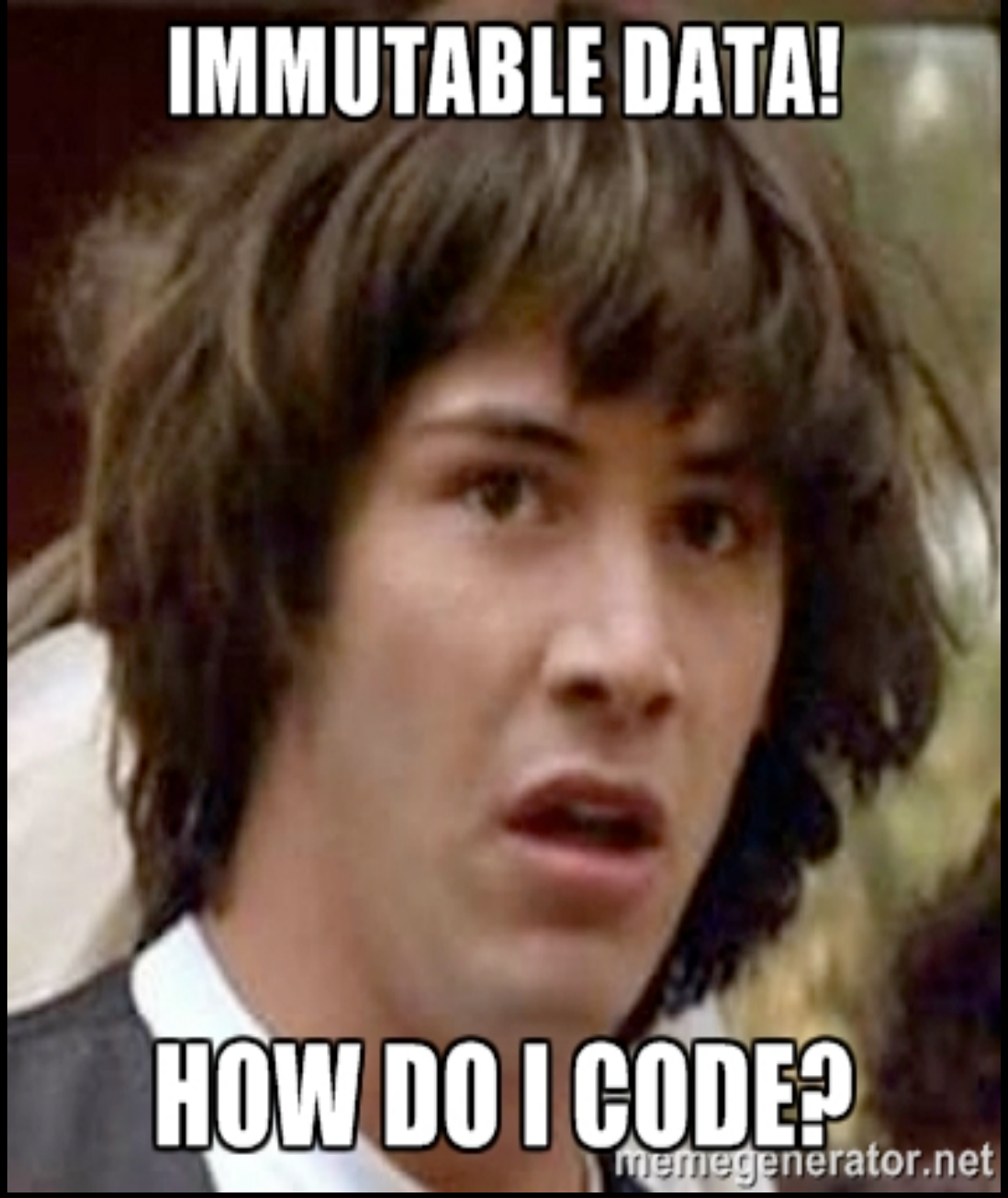
```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

Ereignisorientierte Simulation



Java-Code

- **nicht von mir**
- Institut für Technische Informatik
  Universität der Bundeswehr München
- Prof. Oliver Rose & Kollegen



Simulationsmodelle

```
public interface Model {
    public String getModelName();
    public Event getStartEvent();
}
```

Ereignisse

```
public class Event {

      private final String name;
      private int priority;
      private List<Transition> transitions;
      private List<StateChange> stateChanges;

      public Event(String name) {
            this.name = name;
            this.priority = 0;
            this.transitions =
                  new ArrayList<Transition>();
            this.stateChanges =
                  new ArrayList<StateChange>();
      }
       ...
}
```

Ereignisse

```
public String getName() {
	return name;
}

public int getPriority() {
	return priority;
}

public void setPriority(int priority) {
	this.priority = priority;
}

public List<Transition> getTransitions() {
	return transitions;
}
public List<StateChange> getStateChanges() {
	return stateChanges;
}
```

Ereignisse

```
public void addTransition
            (Transition transition) {
    this.transitions.add(transition);
}

public void addStateChange
            (StateChange stateChange) {
    this.stateChanges.add(stateChange);
}
```

Übergänge

```
public class Transition {

     private final Event targetEvent;
     private Condition condition;
     private Delay delay;

     public Transition(Event targetEvent) {
          this.targetEvent = targetEvent;
          this.condition =
                new TrueCondition();
          this.delay = new ZeroDelay();
     }
     ...
}
```

Übergänge

```
public Event getTargetEvent() {
	return targetEvent;
}

public Condition getCondition() {
	return condition;
}

public void setCondition(Condition condition) {
	this.condition = condition;
}

public Delay getDelay() {
	return delay;
}

public void setDelay(Delay delay) {
	this.delay = delay;
}
```

Modell-Zustand

```
public class ModelState {
     private Map<String, Object> states;

     public ModelState() {
          this.states = new LinkedHashMap<String, Object>();
     }

     public Map<String, Object>
      getStates() {
          return states;
     }
}
```

Zustandsänderungen

```
public interface StateChange {
    public void
    changeState
    (ModelState modelState);
}
```

Zustandsvariable setzen

```
public class SetValueStateChange
  implements StateChange {

    private final String name;
    private final Long value;

    public SetValueStateChange
            (String name, Long value) {
        this.name = name;
        this.value = value;
    }

    @Override
    public void changeState
                (ModelState modelState) {
        modelState.getStates()
          .put(this.name, this.value);
    }
}
```

Zustandsvariable inkrementieren

```
public class IncrementValueStateChange implements StateChange {

    private String name;
    private long increment;

    public IncrementValueStateChange(String name) {
        this.name = name;
        this.increment = 1;
    }

    public IncrementValueStateChange(String name, long increment) {
        this.name = name;
        this.increment = increment;
    }

    @Override
    public void changeState(ModelState modelState) {
        modelState.getStates()
          .put(name,
               ((Long) modelState.getStates().get(name) + increment));
    }
}
```

Verzögerungen

```
public interface Delay {
    public Long getDelay();

}
```

Sofort

```
public final class ZeroDelay
   implements Delay {
      public Long getDelay() {
          return 0L;
      }
}
```

Feste Verzögerung

```
public class ConstantDelay
   implements Delay {

     private final Long value;

     public ConstantDelay(Long value) {
          this.value = value;
     }

     public Long getDelay() {
          return this.value;
     }
}
```

Zufällige Verzögerungen

```
public class ExponentialDelay implements Delay {

	private final double mean;
	private final Random random;

	public ExponentialDelay(double mean,
	                        Random random) {
		this.mean = mean;
		this.random = random;
	}

	public Long getDelay() {
		Long result = 0l;
		double u = random.nextDouble();
		double x = -mean * Math.log(u);
		result = Math.round(x);
		return result;
	}

}
```

Bedingungen

```
public interface Condition {
    public boolean
    isTrue(ModelState modelState);
}
```

Immer

```
public final class TrueCondition
   implements Condition {
     public boolean
      isTrue(ModelState modelState) {
         return true;
     }
}
```

Zustandsabhängig

```
public class LargerThanValueCondition
  implements Condition {

    private String name;
    private Long value;

    public LargerThanValueCondition
           (String name, Long value) {
        this.name = name;
        this.value = value;
    }

    public boolean isTrue(ModelState modelState) {
        return (Long)
           modelState.getStates().get(this.name) >
           this.value;
    }
}
```

- seit 1990 entwickelt
- rein funktional
- fortschrittliches statisches Typsystem
- „lazy evaluation“
- Compiler: ghc

Modell-Zustand

```
public class ModelState {
     private Map<String, Object> states;

     public ModelState() {
          this.states = new LinkedHashMap<String, Object>();
     }

     public Map<String, Object>
      getStates() {
          return states;
     }
}
```

Haskell ist typsicher

```
import qualified Data.Map.Strict
  as Map
import Data.Map.Strict (Map)

type ModelState v = Map String v
```

Typvariable

Modell

```
data Model v = Model {
  modelName :: String,
  startEvent :: Event v
  }
```

Ereignis

```
data Event v = Event {
  name :: String,
  priority :: Int,
  transitions :: [Transition v],
  stateChanges :: [StateChange v]
}
```

Übergang

```
data Transition v = Transition {
      targetEvent :: Event v,
      condition :: Condition v,
      delay :: Delay
}
```

Zustandsänderungen

```
public interface StateChange {
    public void
    changeState
    (ModelState modelState);
}
```

Zustandsänderung

```
import qualified Control.Monad.State.Strict
  as State
import Control.Monad.State.Strict (State)

type ModelAction v a =
  State.State (ModelState v) a

type StateChange v = ModelAction v ()
```

kann nur auf Modell-Zustand operieren

„void“

Zustandsmonade

```
setValue ::
  String -> v -> StateChange v

setValue name value =
  do ms <- State.get
     State.put
         (Map.insert name value ms)
     ms' <- State.get
     return ()
```

Zustandsvariable setzen

```
State.get :: State s s
State.put :: s -> State s ()
type State s a = s -> (s, a)
get :: s -> (s, s)
get st = (st, st)
put :: s -> s -> (s, ())
put newSt oldSt = (newSt, ())
```

Zustandsvariable inkrementieren

Typklasse

```
incrementValue :: Num v =>
   String -> v -> StateChange v

incrementValue name inc =
  do ms <- State.get
     let (Just v) =
           Map.lookup name ms
     setValue name (v + inc)
```

partielle Funktion

Verzögerung

```
import qualified
Control.Monad.Random as Random

type Random =
  Random.Rand Random.StdGen

type Delay = Random Integer
```

Berechnung, die Zufallszahlen benötigt

Verzögerungen

```
zeroDelay :: Delay
zeroDelay = return 0

constantDelay :: Integer -> Delay
constantDelay v = return v

exponentialDelay :: Double -> Delay
exponentialDelay mean =
  do u <- Random.getRandom
     return (round (-mean * log u))
```

Monaden

```
return :: a -> m a


bind ::
   m a -> (a -> m b)
       -> m b
```

Monaden

```
of :: a -> m a
```

```
flatMap ::
   m a -> (a -> m b)
       -> m b
```

Bedingungen

```
type Condition v = ModelState v -> Bool

trueCondition :: Condition v
trueCondition _ = True

largerThanValueCondition :: Ord a =>
   String -> a -> Condition a
largerThanValueCondition name value ms =
  let (Just value') = Map.lookup name ms
  in  value' > value
```

partielle Funktion

Java vs. Haskell

- Haskell viel kürzer
- kein `Object` in Haskell
- keine Zuweisungen in Haskell
- stattdessen Monaden mit expliziten Effekten
- mögliche Effekte in den Typen sichtbar

Weiter geht´s

```
public class EventInstance implements Comparable<EventInstance> {

        private final Event event;
        private final Long time;

        protected EventInstance(Long time, Event event) {
                this.event = event;
                this.time = time;
        }

        protected Event getEvent() {
                return event;
        }

        protected Long getTime() {
                return time;
        }

        public int compareTo(EventInstance eventInstance) {
            ...
        }
}
```

... und wieder Haskell

```
data EventInstance v =
    EventInstance Time (Event v)
  deriving (Show, Eq)

instance Ord (EventInstance v) where
  compare (EventInstance t1 e1)
          (EventInstance t2 e2) = ...
```

Ereignis-Liste

```
public class EventList {

	private List<EventInstance> eventList;

	public EventList() {
		this.eventList = new ArrayList<EventInstance>();
	}

	public void addEvent(EventInstance eventinstance) {
		this.eventList.add(eventinstance);
	}

	public EventInstance removeNextEvent() {
		Collections.sort(this.eventList);
		return this.eventList.remove(0);
	}

	public int getSize() {
		return this.eventList.size();
	}
}
```

Ereignis-Liste

```
import qualified Data.Heap as Heap
import Data.Heap (MinHeap)

... MinHeap (EventInstance v) ...
```

Neue Ereignisse

```
removeNextEvent ::
      Simulation r v (EventInstance v)
removeNextEvent =
  do ss <- State.get
     case Heap.view (events ss) of
       Just (ev, evs') ->
         do State.put (ss { events = evs' })
            return ev
       Nothing -> fail "can't happen"
```

Uhrzeit

```
public class Clock {

	private Long time;

	public Clock(Long time) {
		this.time = time;
	}

	public Long getCurrentTime() {
		return time;
	}

	public void setCurrentTime(Long currentTime) {
		this.time = currentTime;
	}
}
```

Uhrzeit

```
newtype Clock =
  Clock { getCurrentTime :: Time }
```

Hauptprogramm

```
public class MainProgram {

    private Model model;
    private ModelState modelState;
    private EventList eventList;
    private Clock clock;
    private ReportGenerator
            reportGenerator;

    ...
}
```

Simulationszustand

- **`ModelAction`**
- **`Random`**
- + Uhrzeit
- + Ereignis-Liste
- + Report-Generator

Simulationszustand

```
data SimulationState r v =
SimulationState {
  clock :: Clock,
  events :: MinHeap (EventInstance v),
  reportGenerator :: r,
  modelState :: ModelState v,
  randomGenerator :: Random.StdGen
}
```

Simulation

```
type Simulation r v =
  State.State (SimulationState r v)
```

Zeitschritt

```
private EventInstance timingRoutine() {
    EventInstance result =
      this.eventList.removeNextEvent();

    this.clock
      .setCurrentTime(result.getTime());

    return result;
}
```

Zeitschritt

```
timingRoutine ::
  Simulation r v (EventInstance v)
timingRoutine =
  do result <- removeNextEvent
     let (EventInstance t e) = result
     setCurrentTime t
     return result
```

Neue Ereignisse

```
private void eventRoutine(EventInstance eventInstance) {

    Event event = eventInstance.getEvent();
    for (StateChange stateChange : event.getStateChanges()) {
        stateChange.changeState(this.modelState);
    }

    this.reportGenerator.update
       (eventInstance.getTime(), this.modelState);

    for (Transition transition : event.getTransitions()) {

        if (transition.getCondition().isTrue(this.modelState)) {
            Event targetEvent = transition.getTargetEvent();
            Delay delay = transition.getDelay();

            this.eventList.addEvent
              (new EventInstance(this.clock.getCurrentTime()
                                        + delay.getDelay(),
                                 targetEvent));
        }
    }
}
```

Neue Ereignisse

```
generateEvents :: EventInstance v -> Simulation r v ()
generateEvents (EventInstance _ ev) =
  mapM_ (\ tr ->
          do ss <- State.get
             let ms = modelState ss
             if condition tr ms then
               do ss <- State.get
                  let (d, rg) = Random.runRand (delay tr)
                                               (randomGenerator ss)
                  let evi = EventInstance ((getCurrentTime (clock ss)) + d)
                                          (targetEvent tr)
                  let evs' = Heap.insert evi (events ss)
                  State.put (ss { events = evs', randomGenerator = rg })
             else
               return ())
        (transitions ev)
```

Hauptfunktion

```
public void runSimulation(Model model, Long endTime,
                          ReportGenerator reportGenerator)
{
    this.model = model;

    initializationRoutine(reportGenerator);

    while (this.clock.getCurrentTime() <= endTime
           && this.eventList.getSize() > 0) {

        EventInstance currentEvent =
            this.timingRoutine();

        eventRoutine(currentEvent);

    }
}
```

Simulation

```
simulation :: ReportGenerator r v => Time -> Simulation r v ()
simulation endTime =
  let loop =
        do ss <- State.get
           if ((getCurrentTime (clock ss)) <= endTime) &&
              not (Heap.null (events ss)) then
             do currentEvent <- timingRoutine
                updateModelState currentEvent
                updateStatisticalCounters currentEvent
                generateEvents currentEvent
                loop
           else
             return ()
  in loop
```

Modell fortschreiben

```
updateModelState ::
   EventInstance v -> Simulation r v ()

updateModelState (EventInstance _ ev) =
  do ss <- State.get
     let ms = State.execState
                (sequence_ (stateChanges ev))
                (modelState ss)
     State.put (ss { modelState = ms })
```

Initialisierung

```
private void initializationRoutine
      (ReportGenerator reportGenerator) {
    this.clock = new Clock(0L);
    this.modelState = new ModelState();
    this.reportGenerator = reportGenerator;
    this.eventList = new EventList();
    EventInstance initialEvent =
       new EventInstance(
          this.clock.getCurrentTime(),
          this.model.getStartEvent());
    this.eventList.addEvent(initialEvent);
}
```

Alles zusammensetzen

```
runSimulation :: ReportGenerator r v =>
       Simulation r v () -> Model v -> Time -> r -> r
runSimulation sim model clock rg =
  let clock = Clock 0
      initialEvent = EventInstance (getCurrentTime clock)
                                   (startEvent model)
      eventList = Heap.singleton initialEvent
      ss = SimulationState {
             clock = clock,
             events = eventList,
             reportGenerator = rg,
             modelState = Map.empty,
             randomGenerator = mkStdGen 0
           }
      ss' = State.execState sim ss
  in reportGenerator ss'
```

Haskell vs. Java

- anders
- funktional
- stärker getypt
- (lazy)
- Zustandsänderungen über Monaden
- Kombination von Monaden ist Arbeit
- Zustandsänderungen explizit eingegrenzt