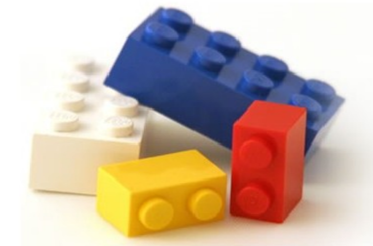
THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

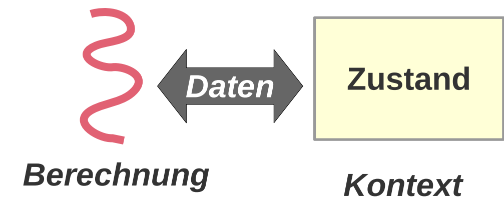


Zustands-Monaden

- State = Reader + Writer
- Beispiele: Ausdrücke mit Seiteneffekten / Postfix-Ausdrücke auswerten

Zustands-Monaden

Reader + Writer = State



Essenz der Reader und Writer

Reader- und Writer-Monaden sind Funktoren (Typ-Konstruktoren) folgender Form:

Reader

Abhängigkeit von einer zu „lesenden“ Datenquelle vom Typ Z

```
Reader[Z, A] ≈ Z => A
```

Als Funktor mit fixiertem Z:

```
ZReader[A] ≈ Z => A
```

```scala
class Reader[Z, A](f: Z => A)

type ZReader[A] = Reader[A, Z]
```

Writer

Abhängigkeit von einer „beschreibbaren“ Datensenke vom Typ W

```
Writer[A, W] ≈ (A, W)
```

Als Funktor mit fixiertem W:

```
WWriter[A] ≈ (A, W)
```

```scala
class Writer[A, W](a: A, w: W)

type WWriter[A] = Writer[A, W]
```

Zustands-Monaden

Reader + Writer = State

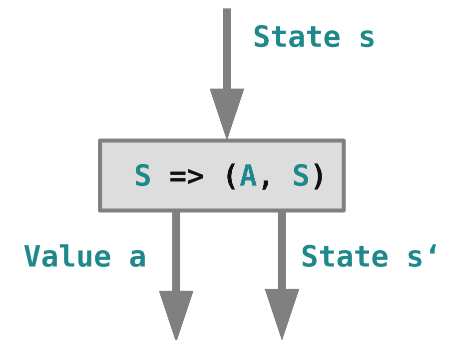
Essenz der State-Monade

State-Monaden sind eine Kombination der Reader- und Writer-Monade

Sie „lesen“ Z-Werte und „schreiben“ W-Werte

Z und W sind dabei der gleiche Typ S: W = Z = S

- `Reader[Z, A] ≈ Z => A`
- `Writer[A, W] ≈ (A, W)`



Kombiniert:

- `State[S, A] ≈ S => (A, S)` `Z = W = S`

Als Funktor mit einem Typ-Argument und fixiertem S:

```
ReaderS[A] ≈ S => A
WriterS[A] ≈ (A, S)
```

Kombiniert:

```
StateS[A] ≈ S => (A, S)
```

```
  class Reader[S, A](f: S => A)
+ class Writer[A, S](a: A, w: S)
--------------------------------------
  class State[A, S](f: S => (A, S))
```

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Einsatz der State-Monade

State-Monaden „lesen“ (konsumieren) S-Werte und „schreiben“ (produzieren) S-Werte

Durch die monadische Herangehensweise sollen mit den S-Werten

- **einfach, elegant und**
- **möglichst im Hintergrund**

umgegangen werden.

Sehr oft sind die S-Werte

- **„funktionale Repräsentanten“ eines veränderlichen Zustands**
- **in dem die Berechnung stattfindet**

Beispiel: Ausdrücke mit Seiteneffekten

Ausdrücke, in denen Variablen vorkommen und deren Auswertung Seiteneffekte haben sind sowohl Reader als auch Writer:

- **Reader: Beachte die Werte der Variablen**
  lies ihren Wert aus dem Kontext
- **Writer: Erzeuge neue Zuordnung Variablen-Name – Variablen-Wert**
  schreibe ihren Wert in den Kontext

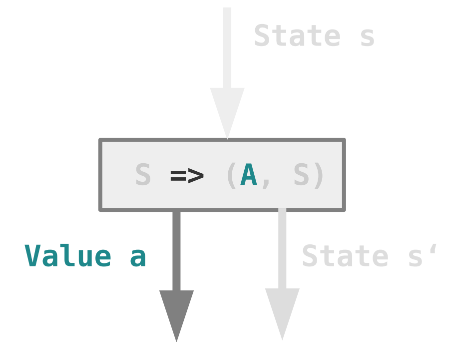
Ausserdem liefern sie noch einen Wert.

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Ausdrücke mit Post-Inkrement Operator (á la C: x++)

Imperative Auswertung: Seiteneffekte und globale Variable



Imperativ: Der Zustand bleibt im Hintergrund

```
enum Term {
  case Literal(v: Int)
  case Variable(name: String)
  case Inc(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}

import Term._

type Env = Map[String, Int]
var globalEnv: Env = Map()

def eval(exp: Term): Int = exp match {
  case Literal(v)   => v
  case Variable(n)  => globalEnv(n)
  case Inc(name)    => // init. 0, post increment
    val v: Int = globalEnv.getOrElse(name, 0)
    globalEnv = globalEnv + (name -> (v+1))
    v
  case Add(t1, t2)  => eval(t1) + eval(t2)
  case Sub(t1, t2)  => eval(t1) - eval(t2)
  case Mult(t1, t2) => eval(t1) * eval(t2)
  case Div(t1, t2)  => eval(t1) / eval(t2)
}
```

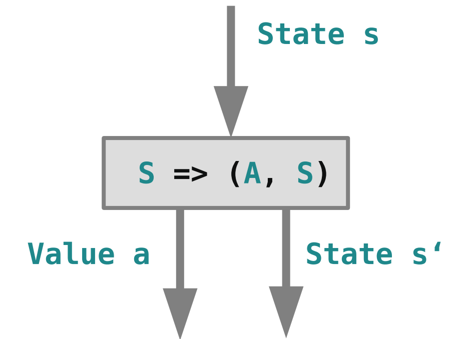
```
val term =
  Add(
    Add(
      Mult(
        Add(Inc("x"), Literal(10)),
        Add(Inc("x"), Inc("x"))),
      Mult(
        Add(
          Inc("y"), Inc("x")),
        Inc("y"))),
    Add(
      Inc("x"),
      Inc("x")))


val result = eval(term) // 42
```

Ein Beispiel

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen



Funktional: Explizite Behandlung des Zustands

Einfache Funktionale Auswertung

Statt Seiteneffekte und globale Variable:
Ein zusätzlicher Parameter (hier gecurryt) und Ergebnisse

```
type Env = Map[String, Int]

def eval(exp: Term): Env => (Int, Env) = exp match {
  case Literal(v)   => env => (v, env)
  case Variable(n)  => env => (env(n), env)
  case Inc(name)    => env => {
    val v: Int = env.getOrElse(name, 0)
    (v, env + (name -> (v+1)))
  }
  case Add(t1, t2)  => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 + v2, env2)
  case Sub(t1, t2)  => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 - v2, env2)
  case Mult(t1, t2) => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 * v2, env2)
  case Div(t1, t2)  => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 / v2, env2)
}
```

```
val term =
  Add(
    Add(
      Mult(
        Add(Inc("x"), Literal(10)),
        Add(Inc("x"), Inc("x"))),
      Mult(
        Add(
          Inc("y"), Inc("x")),
        Inc("y"))),
    Add(
      Inc("x"),
      Inc("x")))


val result = eval(term)(Map())._1 // 42
```

Ein Beispiel

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Einfache Funktionale Auswertung

Statt Seiteneffekte auf globale Variable:

- **zusätzlicher Parameter env (hier gecurryt) und**
- **als Argument (Reader-Verhalten) und weiteres Ergebnis (Writer-Verhalten)**

```
type Env = Map[String, Int]

def eval(exp: Term): Env => (Int, Env) = exp match {

  ...

  case Add(t1, t2)  => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 + v2, env2)

  ...

}
```

Unschöne, nicht-kombinatorische Form:
Der Algorithmus wird auf Werten (v1, v2, env1, env2) formuliert, statt (direkt) auf Funktionen (eval(t1), eval(t2))

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Monadische Form 1: Typ

Beobachtung: eval(term) hat den Typ einer Zustands-Monade

```
type Env = Map[String, Int]

def eval(exp: Term): Env => (Int, Env)
```

Zustands-Monade:

`State[S, A] ≈ S => (A, S)`

```
type Env = Map[String, Int]

type State[S, A] = S => (A, S)

def eval(exp: Term): State[Env, Int]
```

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Monadische Form 2: flatMap / map / pure

Es fehlen noch die passenden Funktionen

```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z => g(f(z)) (z) )
}
```

+

```
class Writer[A, W:Monoid](value: A, w: W) {
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
    val Writer(v, w1) = f(value)
    Writer[B, W](v, Monoid[W].combine(w, w1))
  }
}
```

=

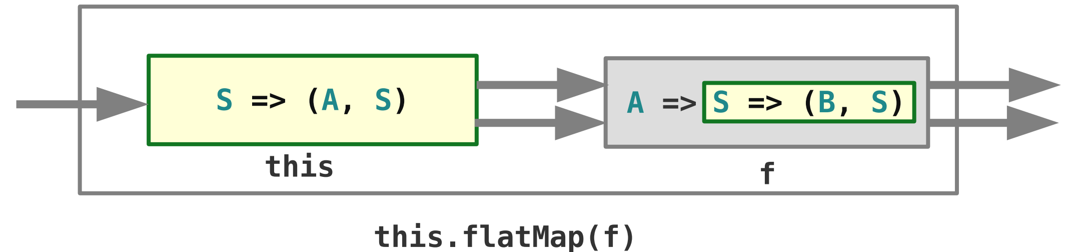
```
class State[A, S](ma: S => (A, S)) {
  ???
}
```

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

3. Monadische Form 2: flatMap / map / pure

Es fehlen noch die passenden Funktionen:



this.flatMap(f)

```
type Env = Map[String, Int]

//type State[S, A] = S => (A, S)

case class State[A, S](ma: S => (A, S)) {
  def apply(s: S) = ma(s)
  def map[B](f: A => B): State[B, S] =
    State(s => {
      val (a, newState) = ma(s)
      (f(a), newState)
    })
  def flatMap[B](f: A => State[B, S]): State[B, S] =
    State(s => {
      val (a, s1) = ma(s)
      f(a)(s1)
    })
}

object State {
  def pure[A, S](a: A): State[A, S] = State(s => (a, s))
}
```

apply: einfache Verwendung als Funktion.

map: wende this.ma an und transformiere den erzeugten Wert mit f.

flatMap: verknüpfe this.ma mit f: wende this.ma an und dann f.
Kombination von
S => (A, S) *und* A => (S => (B => S))

pure: Liefere einen Wert a unabhängig vom Zustand. Verändere den Zustand nicht.

Zustands-Monaden – Beispiel 1

Beispiel: Ausdrücke mit Inc-Operator und Variablen

Ausdrucksauswertung mit For-Comprehension

```
def eval(exp: Term): State[Int, Env] = exp match {
  case Literal(v)   => State.pure(v)
  case Variable(n)  => State(env =>
           (env(n), env))
  case Inc(name)    => State(env => {
           val v: Int = env.getOrElse(name, 0)
           (v, env + (name -> (v+1)))
     })
  case Add(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
    yield v1 + v2
  case Sub(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 - v2
  case Mult(t1, t2) =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 * v2
  case Div(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 / v2
}
```

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Postfix-Ausdrücke

```
enum Operator(op: (Int, Int) => Int) {
  case Add extends Operator( (x, y) => x+y )
  case Sub extends Operator( (x, y) => x-y )
  case Mul extends Operator( (x, y) => x*y )
  case Div extends Operator( (x, y) => x/y )
  def apply(x: Int, y: Int): Int = op(x,y)
}

type OpOrInt = Operator | Int
type PostfixExp = List[OpOrInt]
```

Postfix-Ausdrücke

```
val numPat = """(0|(?:[1-9][0-9]*))""".r
val opPat  = """(\+|\-|\*|/)""".r

def parse(str: String): PostfixExp =
  str.split(" ").toList.map  {
    case numPat(n) =>
      n.toInt
    case opPat(op) =>
      op match {
        case "+" => Operator.Add
        case "-" => Operator.Sub
        case "*" => Operator.Mul
        case "/" => Operator.Div
      }
  }
```

Eine Parsing-Funktion zur vereinfachen Eingabe in Form eines Strings

```
val exp = "1 2 + 3 *" // (1 + 2) * 3

val postfixExp = parse(exp) // List(1, 2, Add, 3, Mul)
```

Ein Beispiel

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Imperative Auswertung

```
class Stack(private var rep: List[Int] = List()) {
  def push(x: Int): Unit =
    rep = x :: rep
  def pop(): Int = rep match {
    case top :: rest =>
      rep = rest
      top
    case _ =>
      throw IllegalStateException("stack is empty")
  }
}
```

Stack: Objekte mit veränderlichemZustand.

```
val stack: Stack = new Stack
```

```
def eval(postFixExp: PostfixExp): Int = postFixExp match {
  case Nil => stack.pop()
  case first :: rest => first match {
    case x: Int =>
      stack.push(x)
      eval(rest)
    case op: Operator =>
      val v1 = stack.pop()
      val v2 = stack.pop()
      stack.push(op(v2,v1))
      eval(rest)
  }
}
```

Seiteneffekte: Die globale Variable vom Typ Stack wird im Hintergrund genutzt.

```
val exp = "4 1 - 3 *" // (4 - 1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp) // 9
```

Beispiel

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Version 1

```
type Stack = List[Int]

def eval(postFixExp: PostfixExp): Stack => Int = postFixExp match {
  case Nil => stack => stack.head
  case first :: rest => first match {
    case x: Int =>
      stack => eval(rest)(x::stack)
    case op: Operator => stack => {
      val v1 = stack.head
      val v2 = stack.tail.head
      eval(rest)(op(v2,v1) :: stack.tail.tail)
    }
  }
}

val exp = "4 1 - 3 *" // (4 - 1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp)(Nil) // 9
```

Eine rein-funktionale Version des imperativen Algorithmus'.

Der veränderliche Stack wird durch einen zusätzlichen Parameter modelliert.

Aber auch wenn der Stack „sich ändert", das alles sieht doch sehr nach Reader aus.

Warum?

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Version 2: Reader

```
class Reader[Z, A](ma: Function1[Z, A]) {
  def this(a: A) = this((ma:Z) => a)
  def apply(z: Z): A = ma(z)
  def map[B](f: A => B) = Reader(ma andThen f)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z => g(ma(z)) (z) )
}

type Stack = List[Int]

def eval(postFixExp: PostfixExp): Reader[Stack, Int] = postFixExp match {
  case Nil => Reader(stack => stack.head)
  case first :: rest => first match {
    case x: Int =>
      Reader(stack => eval(rest)(x::stack))
    case op: Operator => Reader(stack => {
      val v1 = stack.head
      val v2 = stack.tail.head
      eval(rest)(op(v2,v1) :: stack.tail.tail)
    })
  }
}

val exp = "4 1 - 3 *" // (4 - 1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp)(Nil) // 9
```

In der Tat: es handelt sich um einen Algorithmus mit Reader-Signatur.

Ist es ein „Reader-Algorithmus“?

Hmm – flatMap wird nicht benutzt ?

Sagt uns das etwas?

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Version 2: Reader – Gedanken

Auswertung von Postfix-Ausdrücken, vs Auswertung normaler Ausdrücke

```
def eval(postFixExp: PostfixExp): Reader[Stack, Int] =
  postFixExp match {
    case Nil => Reader(stack => stack.head)
    case first :: rest => first match {
      case x: Int =>
        Reader(stack => eval(rest)(x::stack))
      case op: Operator => Reader(stack => {
        val v1 = stack.head
        val v2 = stack.tail.head
        eval(rest)(op(v2,v1) :: stack.tail.tail)
      })
  }
}
```

Beobachtung:

1. Die Auswertung von Ausdrücken in Postfix-Form ist end-rekursiv.

2: Der Mechanismus zur Verkettung / Kombination von Auswertungs-Funktionen (mit flatMap) wird nicht benötigt.

```
def eval(term: Term): Reader[Env, Int] =
  term match {
    case Literal(v) => Reader(v)
    case Const(n) => Reader(env => env(n))
    case Add(t1, t2) =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
      yield v1 + v2
    case Sub(t1, t2) =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 - v2
    case Mult(t1, t2) =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 * v2
    case Div(t1, t2) =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 / v2
  }
```

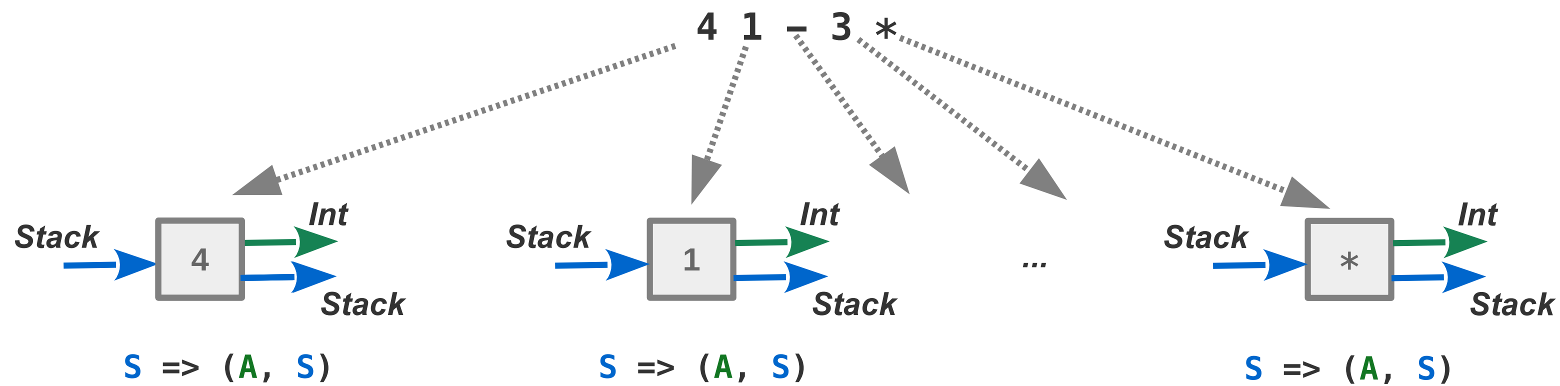
Zum Vergleich

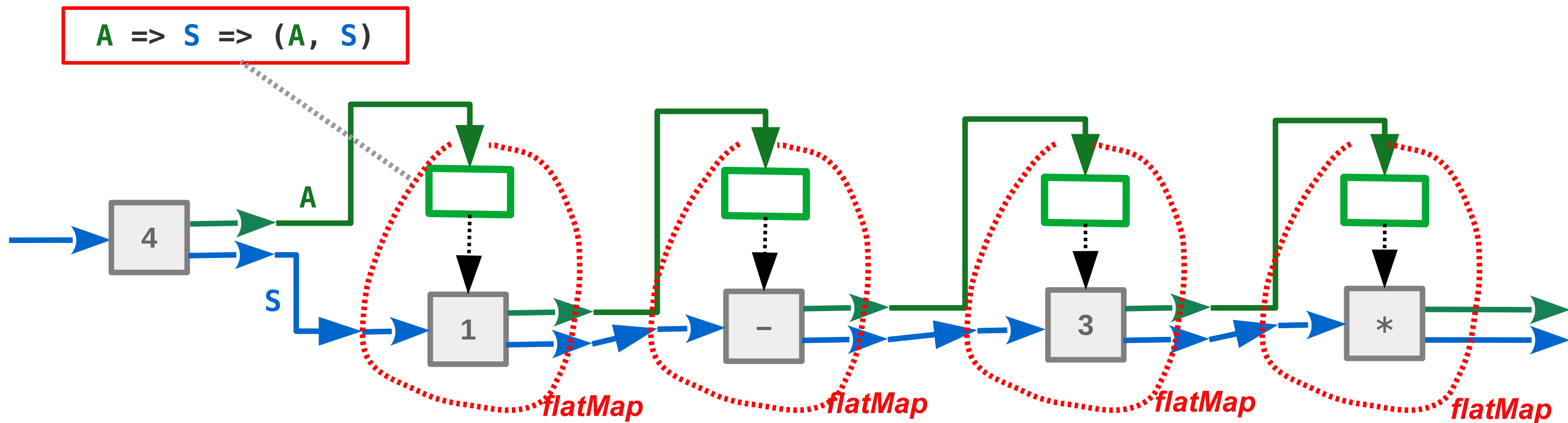
Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Kombinatorische Sicht

eval: Ausdruck (Folge von Symbolen) => Folge von Funktionen, die zu einer kombiniert werden

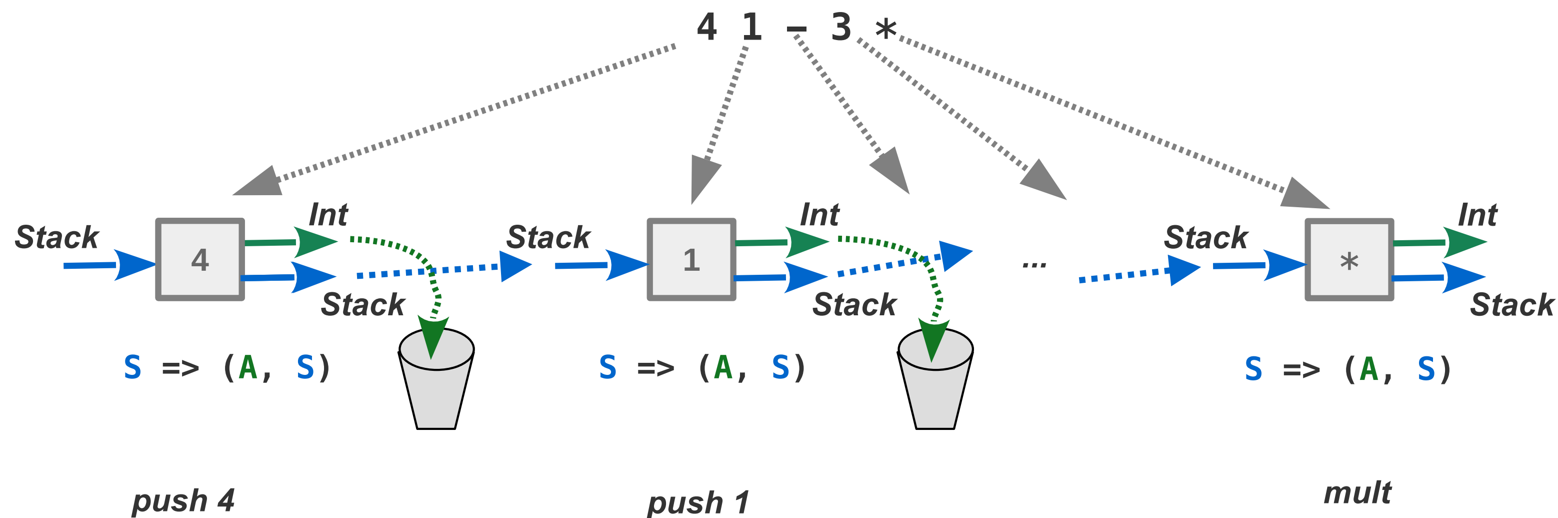




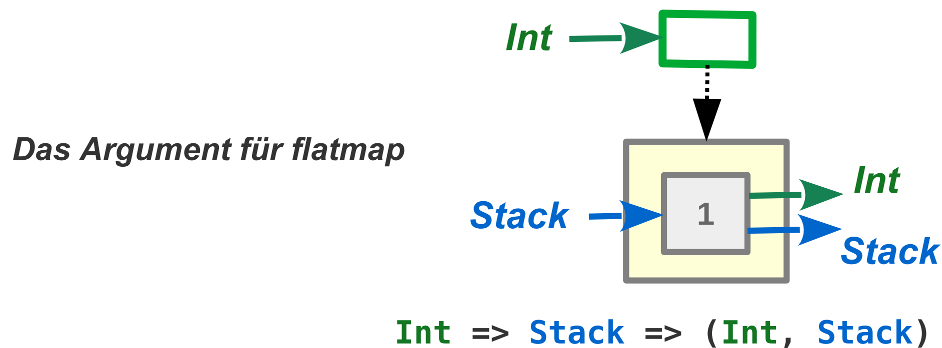
Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

3. Funktionale Auswertung – Kombinatorische Sicht

eval: Ausdruck (Folge von Symbolen) => Folge von Funktionen, die zu einer kombiniert werden



Das Argument für flatmap



flatMap erwartet Argumente vom Typ A => S => (A, S), aber das A-Argument kann ignoriert werden. Die Berechnungen hängen nur vom Stack ab.

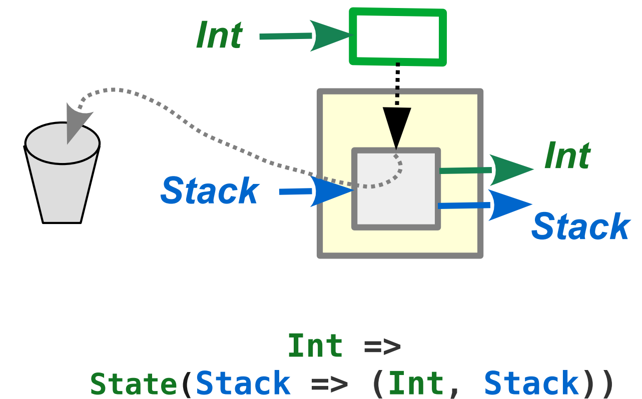
Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

3. Funktionale Auswertung – Kombinatorische Sicht

eval: Ausdruck (Folge von Symbolen) => Folge von Funktionen, die zu einer kombiniert werden

```
def processSymbol(symbol: OpOrInt): Int => StackState[Int] =
  symbol match {
    case x: Int =>
      ignore => State(stack => (x, x :: stack))
    case op: Operator =>
      ignore => State(
        stack => {
          val v1 = stack.head
          val v2 = stack.tail.head
          (op(v2, v1), op(v2, v1) :: stack.tail.tail)
        })
  }
```

Aus einem Symbol ein Argument für flatMap erzeugen



Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Kombinatorische Sicht

eval: Ausdruck (Folge von Symbolen) => Folge von Funktionen, die zu einer kombiniert werden

```
val expValInState = for (
  x <- processSymbol(4)(0);
  y <- processSymbol(1)(x);
  z <- processSymbol(Operator.Sub)(y);
  u <- processSymbol(3)(z);
  v <- processSymbol(Operator.Mul)(u)
) yield v

val expVal = expValInState(Nil) // (9,List(9))
```

Die berechneten Werte werden weitergegeben, aber alle, ausser dem letzten werden ignoriert

Auswertung eines Ausdrucks

Zustands-Monaden – Beispiel 2

Beispiel: Postfix-Ausdrücke mit einem Stack auswerten

Funktionale Auswertung – Kombinatorische Sicht

eval: Ausdruck (Folge von Symbolen) => Folge von Funktionen, die zu einer kombiniert werden

```
def eval(postFixExp: PostfixExp): StackState[Int] = {

  def processSymbol(symbol: OpOrInt): Int => StackState[Int] =
    symbol match {
      case x: Int =>
        _ => State(stack => (x, x :: stack))
      case op: Operator =>
        _ => State(
          stack => {
            val v1 = stack.head
            val v2 = stack.tail.head
            (op(v2, v1), op(v2, v1) :: stack.tail.tail)
          })
    }

  postfixExp.foldLeft( State.pure(0): StackState[Int] )(
    (stackState, symbol) =>
      stackState.flatMap(processSymbol(symbol) )
  )
}

val exp = "4 1 - 3 *" // (4 - 1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp)(Nil) // (9,List(9))
```

Auswertung eines Postfix-Ausdrucks mit einer „Schleife“