Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Ana, Kata, Hylo: Algorithmische Abstraktion

- Kata-, Ana-, und Hylomorphismen
- Tic Tac Toe: MinMax-Algorithmus als Hylomorphismus

Katamorphismen

Katamorphismus

„Kata“: wie „Katastrophe“ κατά στρέφω , hinab stürzen

Funktion die eine Struktur in einen einfachen Wert verwandelt:

Transformiere (falte) eine Datenstruktur zu einemWert.

Wikipedia: Listen-Katamorphismen

Katamorphismen [Bearbeiten | Quelltext bearbeiten]

Sei B ein Datentyp, $b \in B$ ein Wert und $\otimes\colon A \times B \to B$ eine Abbildung. Dann ist durch

$$\begin{aligned} h\colon A^* &\to B \\ Nil &\mapsto b \\ Cons(a, L) &\mapsto a \otimes h(L) \end{aligned}$$

der Katamorphismus (zu griech. κατά = entlang, herab) mit Inititialwert b und Verknüpfung $\otimes$ gegeben.

aus: https://de.wikipedia.org/wiki/Funktionale_Programmierung

Katamorphismus: Strukturelle Rekursion als Faltung

Strukturelle Rekursion: Rekursion über die Struktur der Daten

Beispiel (Generisches) Falten einer Liste:

```
enum IntList {
  case Empty
  case Cons(head: Int, tail: IntList)
}

import IntList.{Empty, Cons}

def foldList[A](f_empty: A,              // Wert für Empty
                f_cons: (Int, A) => A    // Funktion für Cons
               ) : IntList => A = {
  case Empty => f_empty
  case Cons(h, t) => f_cons(h, foldList(f_empty, f_cons)(t))
}

val sumF = foldList[Int](0, {case (i, a) => i + a})

val lst = Cons(1, Cons(2, Cons(3, Empty)))
val sum = sumF(lst) // 6
```

Allgemeines (generisches) Falten

Summe als Instanz der Faltung

Katamorphismus: Strukturelle Rekursion als Faltung

Strukturelle Rekursion: Rekursion über die Struktur der Daten

Beispiel (Generisches) Falten eines Baums:

Datenstruktur

```
enum BTree[+A] {
  case Leaf(v: A)
  case Node(left: BTree[A], right: BTree[A])
}
import BTree.{Leaf, Node}
```

Allgemeines Falten

```
def foldTree[A, B](f_Leaf: A => B,
                   f_Node: (B, B) => B) : BTree[A] => B = {
  case Leaf(v) =>
    f_Leaf(v)
  case Node(l, r) =>
    f_Node(
      foldTree(f_Leaf, f_Node)(l),
      foldTree(f_Leaf, f_Node)(r))
}
```

Summe als Instanz der Faltung

```
val treeSum: BTree[Int] => Int =
  foldTree[Int, Int](
    v => v,
    _ + _
  )

val tree: BTree[Int] = Node(Node(Leaf(1), Leaf(2)), Node(Leaf(3), Leaf(4)))
val treeFolded = treeSum(tree) // 10
```

Anamorphismen / Unfold

Unfold

- Transformiere (Entfalte) einen Wert in eine (zu einer) Datenstruktur
- Ein *Anamorphismus* (von griechisch ana: hinauf)*
- Idee: Nimm
  - einen Startwert (*seed*) s: S,
  - eine Generator-Funktion / (*generator*) g: S → Option(<T, S>) die eventuell ein Paar liefert

  und erzeuge damit eine Folge / einen Strom von T-Werten

* siehe https://en.wikipedia.org/wiki/Anamorphism

Anamorphismen / Unfold

Unfold

Beispiel: Entfalte einer Int-Wert zu einer Int-Liste:

```
def unfold(s: Int)(g: Int => Option[(Int, Int)]): List[Int] =  g(s) match {
  case None          => Nil
  case Some((t, s1)) => t :: unfold(s1)(g)
}

val natsTo10 =
  unfold(1)(x =>
    if (x<10) Some((x, x+1))
    else None)  // List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Unfold / Anamorphismus

Beispiel Entfalte zu einer Liste der Fakultäten mit generischem unfold:

```
def unfold[T](s: T)(g: T => Option[(T, T)]): List[T] =
  g(s) match {
    case None => Nil
    case Some((t, s1)) => t :: unfold(s1)(g)
  }
```

```
def facsTo(n: Int) = unfold[(Int,Int)]((1,0))(
  { case (r, x) =>
    if (x<=n)
      Some(((r, x), (r*(x+1), x+1) ))
    else
      None
  }
)

val facsTo10 = facsTo(10)
// List((1,0), (1,1), (2,2), (6,3), (24,4), (120,5), (720,6), (5040,7), (40320,8), (362880,9), (3628800,10))
```

Unfold / Anamorphismus

Eine Variante des Anamorphismus' / Unfold ist

```
def unfold[A,B](p: B => Boolean, g: B => (A, B)): B => List[A] = {
  def h(b:B): List[A] =
    if (p(b)) Nil
    else {
      val (a, b1) = g(b)
      a :: h(b1)
    }
  h
}

def numbersDownFrom(n: Int) =
  unfold[Int, Int]( (x => x < 0), (x => (x, x-1)) )(n)

def numbersUpTo(n:Int) = unfold[Int, Int]((x => x > n), (x => (x, x+1)) )(0)

val downFrom5 = numbersDownFrom(5) // List(5, 4, 3, 2, 1, 0)
val upTo5 = numbersUpTo(5) //  List(0, 1, 2, 3, 4, 5)
```

Unfold: Bäume entfalten

Unfold kann von Listen auf andere Datenstrukturen verallgemeinert werden

Beispielsweise auf (Binär-) Bäume:

```
enum Tree[+T] {
  case Leaf(v: T)
  case Node(v: T, l: Tree[T], r: Tree[T])
}
import Tree.{Leaf, Node}

def unfold[A,B]( p:  B => Boolean,  // stop
                 f:  B => A,        // create node value
                 g1: B => B,        // seed for left branch
                 g2: B => B         // seed for right branch
               ): B => Tree[A] = {

  def h(b:B): Tree[A] =
    if (p(b)) Leaf(f(b))
    else {
      val b1 = g1(b)
      val b2 = g2(b)
      Node(f(b), h(b1), h(b2))
    }

  h
}
```

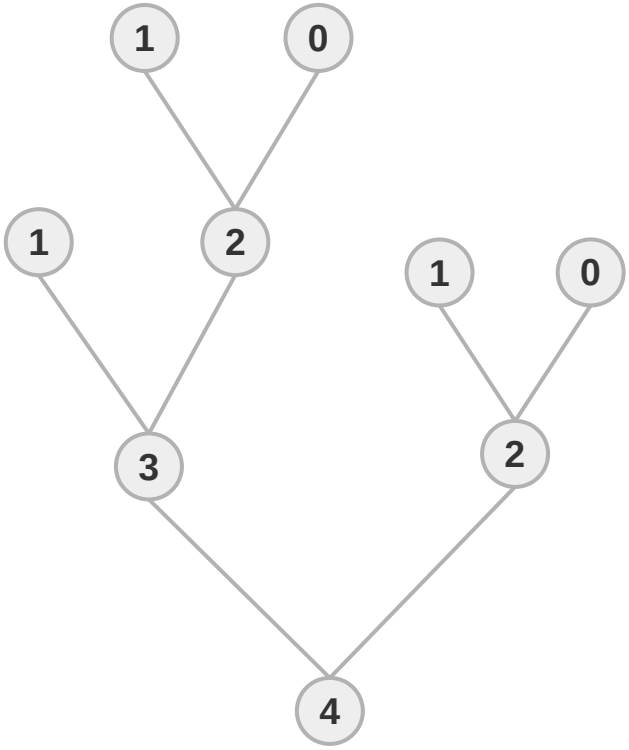
Unfold: Bäume entfalten

Unfold kann von Listen auf andere Datenstrukturen verallgemeinert werden

Der Aufrufbaum der Fibonacci-Funktion kann mit diesem unfold erzeugt werden

```
def fibCallTree(n: Int) = unfold[Int, Int](
  x => x <= 1,
  x => x,
  x => x-1,
  x => x-2)(n)

val fibCallTree4 = fibCallTree(4)
```

```
Node(4,
     Node(3,
          Node(2,
               Leaf(1),
               Leaf(0)
              ),
          Leaf(1)
     ),
     Node(2,
          Leaf(1),
          Leaf(0)
     )
)
```



Hylomorphismus: Unfold + Fold

Die Berechnung der Fakultäten kann in zwei Schritten erfolgen

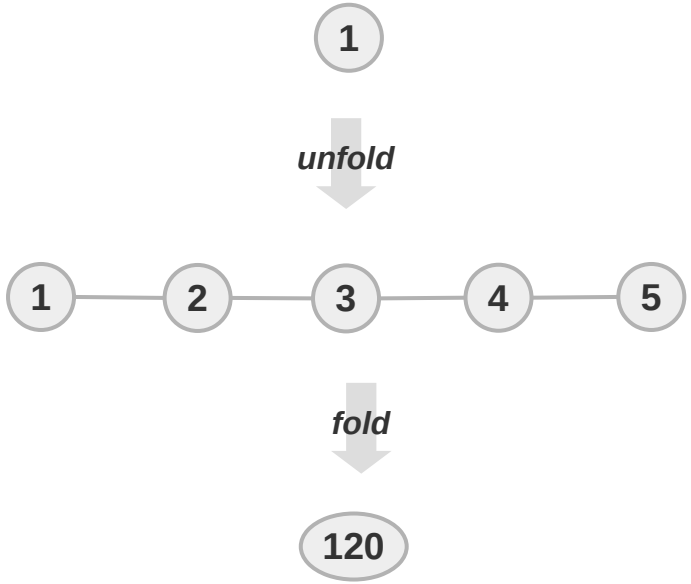
- **Unfold**

  Generiere die Liste [1, 2, 3, 4, 5]

- **Fold**

  Falte die Liste mit der Multiplikation

```
def unfold[A,B](p: B => Boolean, g: B => (A, B)): B => List[A] = {
  def h(b:B): List[A] =
    if (p(b)) Nil
    else {
      val (a, b1) = g(b)
      a :: h(b1)
    }
  h
}

def natsTo(n: Int): List[Int] =
  unfold[Int, Int](
      x => x > n,
      x => (x, x+1)
    )(1)


val f5 = natsTo(5).foldLeft(1)( _ * _) // 120
```



Hylomorphismus: Unfold + Fold

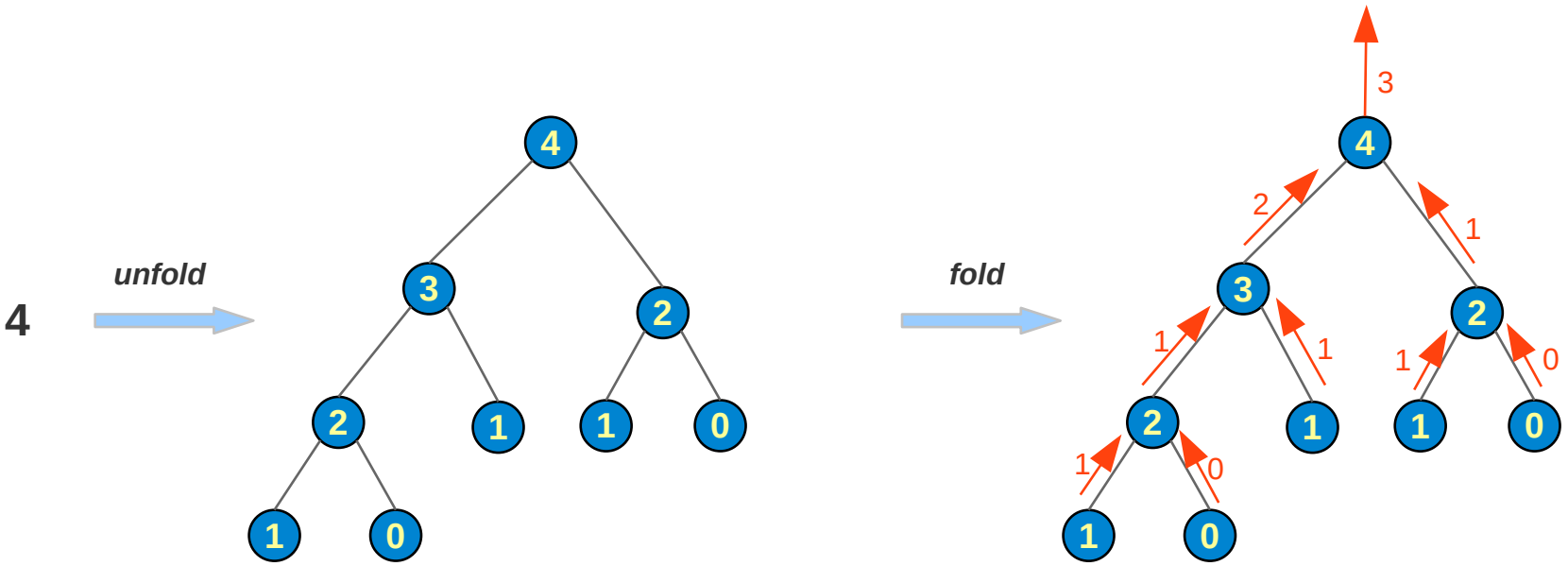
Hylomorphismus

Kombination aus Unfold (Anamorphismus) und Fold (Catamorphismus)

Beispiel Fibonacci-Funktion

Die Fibonacci-Funktion ist ein Hylomorphismus mit

- **Unfold:** Ein Baum mit „Sub-Werten“
- **Fold:** Addition der Sub-Werte



Hylomorphismus: Unfold + Fold

Beispiel Fibonacci-Funktion

```
enum Tree[+T] {
  case Leaf(v: T)
  case Node(v: T, l: Tree[T], r: Tree[T])
}
import Tree.{Leaf, Node}

def unfold[A,B](
                 p:  B => Boolean,
                 f:  B => A,
                 g1: B => B,
                 g2: B => B
               ): B => Tree[A] = {

  def h(b:B): Tree[A] =
    if (p(b)) Leaf(f(b))
    else {
      val b1 = g1(b)
      val b2 = g2(b)
      Node(f(b), h(b1), h(b2))
    }

  h
}
```

```
def fold[A, B](f_Leaf: A => B,
               f_Node: (A, B, B) => B) : Tree[A] => B = {
  case Leaf(v) =>
    f_Leaf(v)
  case Node(v, l, r) =>
    f_Node(
      v,
      fold(f_Leaf, f_Node)(l),
      fold(f_Leaf, f_Node)(r))
}
```

```
def fib(n: Int): Int = {
  val tree = unfold[Int, Int](x => x <= 1, x => x,
                              x => x-1,    x => x-2)(n)
  fold(
    (v: Int) => v,
    (v:Int, x:Int, y:Int) => x + y)(tree)
}
```

```
val fib5 = fib(10) // 55
```

Kata, Ana, Hylo auf Bäumen

Verallgemeinerung auf Bäume mit beliebig vielen Unterbäumen

```
enum Tree[+A] {
  case Empty extends Tree[Nothing]
  case Node(v: A, succs: Seq[Tree[A]])
}

import Tree._

// Katamorphismus
case class TreeKata[IN, OUT](v_empty: OUT, f_node: (IN, Seq[OUT]) => OUT) extends Function[Tree[IN], OUT] {
  override def apply(tree: Tree[IN]): OUT = tree match {
    case Empty => v_empty
    case Node(v, succs) => f_node(v, succs.map( t => apply(t)) )
  }
}

// Anamorphismus
case class TreeAna[IN, OUT](p: IN => Boolean, f: IN => OUT, g: IN => Seq[IN]) extends Function[IN, Tree[OUT]] {
  override def apply(b: IN): Tree[OUT] =
    if (p(b))
      Empty
    else {
      val seeds = g(b)
      Node (f(b), seeds.map( seed => apply(seed)))
    }
}

// Hylomorphismus
case class TreeHylo[IN, M, OUT](kata: TreeKata[M, OUT], ana: TreeAna[IN, M]) extends Function[IN, OUT]{
  override def apply(x: IN): OUT =
    ana.andThen(kata)(x)
}
```

Beispiel: MinMax-Algorithmus

Zwei-Personen-Nullsummen-Spiel

Spiel bei dem der Gewinn des einen der Verlust des anderen ist

Beispiel TickTackToe

Bettspiel für zwei Personen (Weiß und Schwarz)

Vier Spielsteine in einer Reihe (Zeile/Spalte/Diagonale) gewinnen

- **entweder einer gewinnt und der andere verliert** — **Gewinn: 1 bzw. –1**
- **oder es ist unentschieden** — **Gewinn: für beide 0**

X	O	X
O	X	
X	O	

Weiß (Spielstein X) hat gewonnen

Beispiel: MinMax-Algorithmus

MinMax-Algorithmus

Berechnung des nächsten Spielzugs durch Auswertung des Spielbaums

Spielbaum

Ausgehend von der aktuellen Situation (Konfiguration) auf dem Spielfeld werden die Stellungen (Konfigurationen) berechnet die sich aus allen möglichen Zügen und Folgezügen der Spieler ergeben.

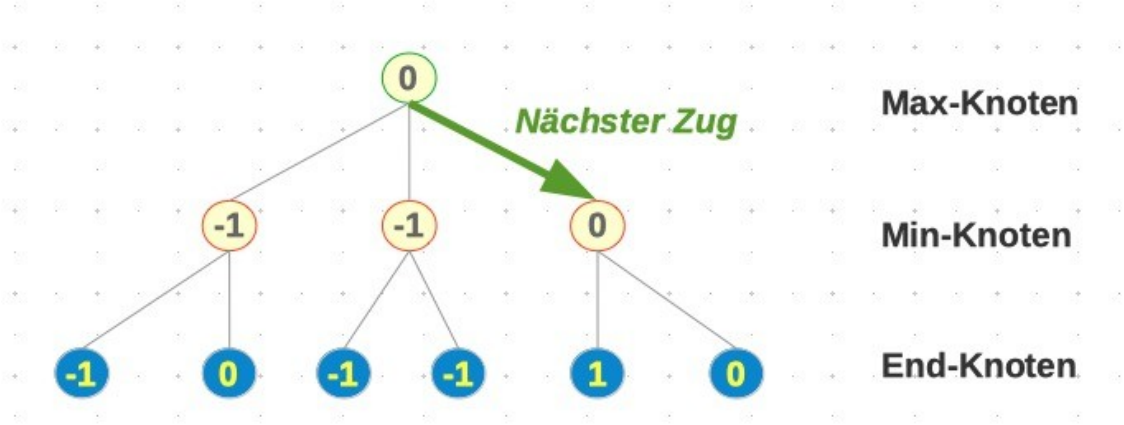
Daraus ergibt sich eine Bewertung der nächsten Züge als

- **Maximum (ich wähle den besten Zug für mich)**
- **der Minima (Gegner wird besten Zug für sich wählen)**

der Bewertungen der Unterbäume.

Endknoten (finale Stellungen) werden mit 1 / -1 / 0 bewertet

(ich gewinne / Gegner gewinnt / unentschieden)



Beispiel: MinMax-Algorithmus

MinMax-Algorithmus – generisch

operiert auf Konfigurationen c (Spielstellungen):

```
minMax(c) =
     score(c)                                          falls c ist Endstellung
     max { minMax(c') | c' ist Folgezustand von c }    falls c ist Max-Knoten (Weiß zieht)
     min { minMax(c') | c' ist Folgezustand von c }    falls c ist Min-Knoten (Schwarz zieht)
```

Beispiel: MinMax-Algorithmus

MinMax-Algorithmus als Hylomorphismus

Bestimmung des besten Spielzugs als Hylomorphismus:

- **Anamorphismus: Baumaufbau**
- **Katamorphismus: Minimax-Algorithmus**

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

MinMax

Algorithmus zur Bestimmung des besten nächsten Zugs für beliebige 2-Personen-Nullsummenspiele.

Spielstand = Konfiguration.

Der Algorithmus sucht die beste Folgekonfiguration durch Suche im Spielbaum

Algebra der Konfigurationen C für den Baumaufbau

- score: C => Int
- isFinal: C => Boolean
- successors: C => C*

Spiel-Bäume

- haben Min- und Max-Knoten
- die jeweils eine Konfiguration enthalten

```
minMax(c) =
     score(c)                                          falls c ist Endstellung
     max { minMax(c') | c' ist Folgezustand von c }    falls c ist Max-Knoten (Weiß zieht)
     min { minMax(c') | c' ist Folgezustand von c }    falls c ist Min-Knoten (Schwarz zieht)
```

MinMax-Algorithmus: Bewertung eines Spielbaums
Knoten und die in ihnen enthaltenen Konfigurationen werden nicht klar unterschieden
(c ist Konfiguration und Knoten mit Konfiguration c)

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

Spieler, Spielsteine und Konfigurationen

```
enum Player {
  case WhitePlayer
  case BlackPlayer
}
```

Es gibt zwei Spieler: Schwarz und Weiß

```
enum TicTacToeToken(val owner: Player) {
  case X extends TicTacToeToken(WhitePlayer)
  case O extends TicTacToeToken(BlackPlayer)
}
```

Es gibt zwei Spielsteine:
– O gehört Schwarz und
– X gehört Weiß

```
type TicTacToe_Config = Array[Array[Option[TicTacToeToken]]]
```

3x3 Matrix aus eventuellen Tokens: naheliegende aber zu speicherineffiziente Datenstruktur für Konfigurationen

MinMax Berechnung für Tick Tack Toe

Konfigurationen (Spielstellungen)

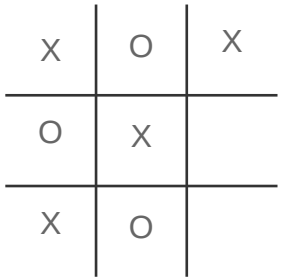
```
class TicTacToeConfig(val configVector: Vector[Int]) {

  val X: Int = 1
  val O: Int = -1

  def score(): Int =
    isWinning() match {
      case Some(WhitePlayer) => 1
      case Some(BlackPlayer) => -1
      case None => 0
    }

  def successors(player: Player): Seq[TicTacToeConfig] = {
    val tok = if (player == WhitePlayer) X else O
    if (isFinal()) Nil
    else for (i <- 0 until 9
          if configVector(i) == 0
        ) yield TicTacToeConfig(configVector.updated(i, tok))
  }

  ...
```

1 steht für Spielstein X
-1 steht für Spielstein O

Bewertung einer Stellung

Nachfolger dieser Stellung falls Spieler **player** *am Zug ist*

Ein Vector der Länge 9 repräsentiert ein 3x3 Spielbrett positionsweise
[(0,0), (0,1), 0,2),
(1,0), (1,1), 1,2),
(2,0), (2,1), 2,2)]



~ *[1, -1, 1,*
-1, 1, 0,
1, -1, 0]

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

Konfigurationen (Spielstellungen)

```
class TicTacToeConfig(val configVector: Vector[Int]) {

  ...


def isWinning(): Option[Player] = {
  val threeInARow: Option[(Int, Int, Int)] = Seq(
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6)).find {
    case (i, j, k) =>
      (configVector(j) == configVector(i)) & (configVector(k) == configVector(i))
  }
  threeInARow match {
    case Some((i, _, _)) if configVector(i) == X =>
      Some(WhitePlayer)
    case Some((i, _, _)) if configVector(i) == 0 =>
      Some(BlackPlayer)
    case _ =>
      None
  }
}
```

Eine Konfiguration ist eine Gewinnerkonfiguration wenn drei gleiche Spielsteine in einer Zeile, Spalte oder Diagonale zu finden sind

Zeilenpositionen: ***(0,1,2), (3,4,5), (6,7,8)***

Spaltenpositionen: ***(0,3,6), (1,4,7), (2,5,8)***

diagonale Positionen: ***(0,4,8), (2,4,6)***

0	1	2
3	4	5
6	7	8

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

Konfigurationen (Spielstellungen)

```
class TicTacToeConfig(val configVector: Vector[Int]) {

  ...


  def isFinal(): Boolean =
    isFull(configVector) || isWinning().isDefined

  def isFull(config: Vector[Int]): Boolean = !config.contains(0)



}
```

Eine Endstellung

Brett ist voll

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

Spielbäume: mögliche Stellungen ab einer bestimmten Stellung

```
enum MinMaxNode(val config: TicTacToeConfig) {
  case MaxNode(override val config: TicTacToeConfig) extends MinMaxNode(config)
  case MinNode(override val config: TicTacToeConfig) extends MinMaxNode(config)
}

import MinMaxNode._

final case class MinMaxTree(
  node: MinMaxNode,
  successors: Seq[MinMaxTree])
```

Knoten: Max oder Minknoten mit Stellung (Konfiguration)

Spielbaum

MinMax Berechnung für Tick Tack Toe

MinMax Algorithmus als Hylomorphismus

```
def bestNextConfigByMinMax(config: TicTacToeConfig, player: Player): TicTacToeConfig = {
  val ana: TreeAna[MinMaxNode, MinMaxNode] =
    TreeAna(
      node => node.config.isFinal(),
      node => node,
      node => node match {
        case MaxNode(c) => c.successors(WhitePlayer).map(c => MinNode(c))
        case MinNode(c) => c.successors(BlackPlayer).map(c => MaxNode(c))
      }
    )
  val kata: TreeKata[MinMaxNode, Int] =
    TreeKata[MinMaxNode, Int](0, (node, ys) =>
      node match {
        case MaxNode(c) => ys.max
        case MinNode(c) => ys.min
      }
    )

  def treeHylo: TreeHylo[MinMaxNode, MinMaxNode, Int] = TreeHylo(kata, ana)

  player match {
    case WhitePlayer =>
      config.successors(WhitePlayer).maxBy(c => treeHylo(MaxNode(c)))
    case BlackPlayer =>
      config.successors(BlackPlayer).minBy(c => treeHylo(MinNode(c)))
  }
```

Der Algorithmus bestimmt die für einen Spieler beste Folge-Konfiguration

Beispiel: MinMax-Algorithmus

MinMax Berechnung für Tick Tack Toe

Test

```
val actConfig: TicTacToeConfig = TicTacToeConfig( Vector(
   1, -1, 0,
  -1,  1, 0,
   1, -1, 0))

val expected: TicTacToeConfig = TicTacToeConfig( Vector(
   1, -1,  1,
  -1,  1,  0,
   1, -1,  0))

val nextConfig = bestNextConfigByMinMax(actConfig, WhitePlayer)

assert(expected.configVector == nextConfig.configVector)
```