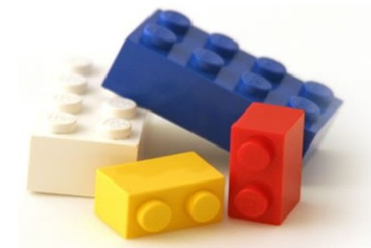
THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Software-Komponenten

Th. Letschert

THM

University of Applied Sciences



Backtracking und die Plus-Monade

- **Backtracking**
- **Plus-Monade**

Nicht-Determinismus und Backtracking

Backtracking und Nichtdeterministische Programme

Nichtdeterminismus als Abstraktion eines Auswahl- / Such-Mechanismus

Wähle

- **jetzt einen (richtigen) unter mehreren Werten, derart, dass sich**
- **später keine Probleme ergeben**

```
teilLösung <~ leereLösung
while ⌐ istFertigeLösung (teilLösung) {
        s <~ Orakel ({s | s ist möglicher nächster Schritt})
        teilLösung = teilLösung + s
}
```

Nichtdeterministischer Algorithmus:
Das Orakel wählt „mit Magie“ den einen nächsten Schritt derart, dass sich später keine Probleme ergeben.

Das ist kein Algorithmus! Bei einem Algorithmus muss der nächste Schritt stets determiniert sein.
Es handelt sich um die Spezifikation eines Such-Algorithmus, bei dem nach der / einer richtigen Folge von Auswahlschritten gesucht wird.

Hinweis: Neben diesem

„don‘t know“-Nichtdeterminismus **: „Ich weiß nicht was die richtige Entscheidung ist“**

gibt es auch noch den

„don‘t care“-Nichtdeterminismus **: „Es ist egal welche Entscheidung getroffen wird“**

Nicht-Determinismus und Backtracking

Backtracking und Nichtdeterministische Programme

Algorithmische Realisation des Nichtdeterminismus

Nichtdeterministischer Algorithmus ~> Suche

Die Suche muss organisiert werden:

- durch den Entwickler, oder
- durch die Implementierung einer Sprache mit entsprechenden Ausdrucksmitteln

Nichtdeterminismus als Sprachmerkmal

Die Sprache liefert eine automatische Realisation der notwendigen Suche

Verbreitete „Nichtdeterministische Sprachen“ sind Logik-Sprachen z.B. Prolog

Choice-Fail: Abstrakte Implementierung des Nichtdeterminismus

Choice – Fail

Nichtdeterminismus kann abstrakt definiert werden mit zwei Operationen:

– `choice` **(*Alternativen*)**

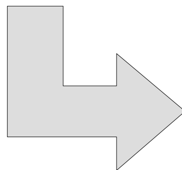
deklariert Entscheidungs-Alternativen als Wert.

– `fail`

bringt zum Ausdruck, dass keine Entscheidungsalternativen mehr zur Verfügung stehen und die Berechnung gescheitert ist.

```
teilLösung ←  leereLösung
while ⌐ istFertigeLösung (teilLösung) {
        s ← Orakel ({s | s ist möglicher nächster Schritt})
        teilLösung = teilLösung + s
}
```

Wird mit choice / fail formuliert als



```
teilLösung ← leereLösung
while ⌐ istFertigeLösung (teilLösung) {
        s ← choice ({s | s ist möglicher nächster Schritt})
        teilLösung = teilLösung + s
        if (nichtOK(teilLösung)) fail
}
```

Choose-Fail: Abstraktion des Backtrackings

Choose – Fail

liefert eine Abstraktion der erschöpfenden Suche mit einem Backtracking-Algorithmus

Backtracking

ist eine Implementierungs-Variante für Choice-/Fail-Nichtdeterminismus:

- `choose` **(*Alternativen*)**
  liefert eine der Entscheidungs-*Alternativen* als Wert.
  Speichert die aktuelle Situation (mit den noch nicht ausgewählten Werten).
- `fail`
  geht zurück zur letzten Auswahl, deren Alternativen noch nicht erschöpft sind.
  Von dort (der gespeicherten Situation bei Auswahl) aus weiter mit der nächsten Wahl.

Nicht-Determinismus und Backtracking

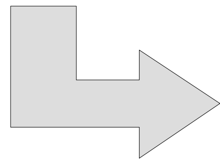
Choose-Fail: Abstraktion des Backtrackings

Mit Backtracking kann der Choose- / Fail- Mechanismus implementiert werden

Der Laufzeitstack dient dabei als Speicher der aktuelle Situation bei einer Auswahl

Dazu muss muss die Auswahl stets mit einem (rekursiven) Funktionsaufruf verbunden werden.

```
teilLösung ← leereLösung
while ⌐ istFertigeLösung (teilLösung) {
        s ← choose ({s | s ist möglicher nächster Schritt})
        teilLösung = teilLösung + s
        if (nichtOK(teilLösung)) fail
}
```



Wird implementiert mit Hilfe des Stacks

```
löse (teilLösung) {
   if istFertigeLösung (teilLösung) {
        return teilLösung
   } else {
        foreach s in {s | s ist möglicher nächster Schritt} {
            if (OK(teilLösung))
               try
                    löse(teilLösung + s)
               catch failure
       }
       throw failure
}
```

Choose-Fail als Programm-Komponente

Choice / Fail als Bestandteile einer generischen Komponente

Neben der Möglichkeit Choice/Fail durch den

- **Sprache-Implementierer, oder**
- **Programm-Implementierer**

zu realisieren,

gibt es noch die Möglichkeit sie als Bestandteile einer

- **Programmkomponente,**
- **eventuell als Teil einer Bibliothek**

zu realisieren.

Nicht-Determinismus generisch

Beispiel Pythagoreische Tripel

Nichtdeterrministisches Programm (-Spezifikation)

Berechnung eines oder aller pythagoreischen Tripel (x, y, z mit $x^2 + y^2 = z^2$)

```
def triple() = {
  val i = choose(2, 3, 4, 5)
  val j = choose(2, 3, 4, 5)
  val k = choose(2, 3, 4, 5)
  if (i*i + j*j != k*k) fail()
  succeed(i, j, k)
}
```

Choice/Fail-Algorithmus

Wähle drei Zahlen i, j, k

Wenn i, j und k kein pythagoreisches Tripel bilden, dann war die Wahl falsch, ansonsten sind sie die Lösung oder eine der Lösungen

Nicht-Determinismus generisch

Nichtdeterminismus: 2 Implementierungen

2 Implementierungen der nichtdeterministischen Programm-Spezifikation

a) Alle Lösungen suchen, mit funktionalem Algorithmus

b) Eine Lösung suchen, mit imperativem Algorithmus

```
type Triple = (Int, Int, Int)

def allPTriples: List[Triple] =
  for (i <- List(1, 2, 3, 4, 5);
       j <- List(1, 2, 3, 4, 5);
       k <- List(1, 2, 3, 4, 5);
       if i * i + j * j == k * k)
    yield (i, j, k)

val pTripleList = allPTriples //List((3,4,5), (4,3,5))
```

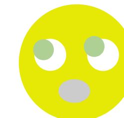
***Funktionale Implementierung*, bestimmt *alle* Lösungen.**
Der Suchraum wird komplett traversiert.

```
def aPTriple_Imp: Triple = {
  for (i <- List(1, 2, 3, 4, 5))
    for (j <- List(1, 2, 3, 4, 5))
      for (k <- List(1, 2, 3, 4, 5)) {
        if (i * i + j * j == k * k)
          return (i, j, k)
      }
  throw new Exception("no triple found")
}

val pTriple = aPTriple_Imp //(3,4,5)
```

***Imperative Implementierung*, bestimmt *eine* Lösung.**
Der Suchraum wird solange traversiert, bis eine Lösung gefunden wurde, oder alle Kandidaten untersucht sind.

Ganz und gar imperativ:
- ***Konrolltransfer mit return und throw***
- ***Imperative Schleifen***



Nicht-Determinismus generisch

Nichtdeterminismus – generisch

Backtracking bzw. **Lösungsmenge suchen**
sind jeweils nur **eine Möglichkeit Nichtdeterminismus** zu **implementieren**

Generischer ND Algorithmus:

- ND Algorithmus ohne Angabe der Implementierung
- Spezifiziert mit den Kern-Mechanismen des Nichtdeterminismus:
  - **Auswahl**: choose
  - **Erfolg**: succeed
  - **Fehlschlag**: fail
- Mit den beiden **Implementierungen** (Lösungsmenge / Backtracking) als **Instanzen** des generischen Algorithmus

```
def triple() = {
  val i = choose(2, 3, 4, 5)
  val j = choose(2, 3, 4, 5)
  val k = choose(2, 3, 4, 5)
  if (i*i + j*j != k*k) fail()
  succeed(i, j, k)
}
```

Alle Lösungen

```
def triple: List[Triple] =
  for (i <- List(1, 2, 3, 4, 5);
       j <- List(1, 2, 3, 4, 5);
       k <- List(1, 2, 3, 4, 5);
       if i * i + j * j == k * k)
    yield (i, j, k)
```

Eine Lösung

```
def triple: Triple = {
  for (i <- List(1, 2, 3, 4, 5))
    for (j <- List(1, 2, 3, 4, 5))
      for (k <- List(1, 2, 3, 4, 5)) {
        if (i * i + j * j == k * k)
          return (i, j, k)
      }
  throw new Exception("no triple found")
}
```

Hier muss natürlich noch alles Imperative vertrieben werden !

Nichtdeterminismus – generisch: MonadPlus

Die Mechanismen des Nichtdeterminismus:

- **Auswahl: choose**
- **Erfolg: succeed**
- **Fehlschlag: fail**

können in einer Typklasse `MonadPlus` gekapselt werden

Suchen nach einer Lösung : (funktionales !) Backtracking
sollte als Instanz von `MonadPlus` definierbar sein

Suchen nach allen Lösungen : funktionaler Algorithmus zur Bestimmung aller Lösungen
sollte als Instanz von `MonadPlus` definierbar sein

```
type Triple = (Int, Int, Int)

def triple[M[_]: MonadPlus](): M[Triple] =
  for ( i <- MonadPlus[M].choose(2, 3, 4, 5);
        j <- MonadPlus[M].choose(2, 3, 4, 5);
        k <- MonadPlus[M].choose(2, 3, 4, 5);
        r <- (if (i*i + j*j == k*k) MonadPlus[M].succeed(Tuple3(i, j, k)) else MonadPlus[M].fail))
    yield r
```

Nichtdeterminismus – generisch: MonadPlus

MonadPlus muss eine Monade sein

und zudem noch fail / succeed und choose definieren:

```
trait Functor[F[_]] {
  extension[A, B] (fa: F[A]) {
    def map(f: A => B): F[B]
  }
}

trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  extension[A, B] (x: F[A]) {
    def flatMap(f: A => F[B]): F[B]
    def map(f: A => B) = x.flatMap(f.andThen(pure))
  }
}
object Monad {
  def apply[F[_]: Monad] = summon[Monad[F]]
}
```

```
trait MonadPlus[M[_]] extends Monad[M] {
  def fail[A]: M[A]
  def succeed[A](a:A) = pure(a)
  def choose[A](alternatives: A*): M[A] = ???
}
object MonadPlus {
  def apply[M[_]: MonadPlus] = summon[MonadPlus[M]]
}
```

Nichtdeterminismus – generisch: MonadPlus

MonadPlus

Eine Monade mit plus (~ +) und fail (~ 0)

```
trait MonadPlus[M[_]] extends Monad[M] {
  def fail[A]: M[A]
  def succeed[A](a:A) = pure(a)
  def choose[A](alternatives: List[A]): M[A] =
    alternatives.foldLeft(fail[A])(
      (acc, i) => acc plus pure(i)
    )
  def choose[A](alternatives: A*): M[A] =
    choose(alternatives.toList)

  extension[A, B] (x: M[A]) {
    def plus(y: M[A]): M[A]
  }
}
object MonadPlus {
  def apply[M[_]: MonadPlus] = summon[MonadPlus[M]]
}
```

succeed *nimmt einen Wert und verpackt ihn in der Monade, dazu haben wir schon ein* pure*.*

choose *wählt einen aus vielen Werten, wir reduzieren es auf eine Basis-Operation* plus*. Die zweite Variante dient der Bequemlichkeit beim Aufruf.*

plus *wählt zwischen zwei Werten. Die letzte Wahl, wenn nichts mehr zu wählen gibt, ist natürlich fail.*

Nicht-Determinismus generisch

List als MonadPlus

Instanzierung von MonadPlus mit List

Listen sind eine recht offensichtliche Instanz von MonadPlus

```
given MonadPlus[List] with {
  def pure[A](x: A): List[A] = List(x)
  def fail[A]: List[A] = Nil
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] = xs.flatMap(f)
    override def map(f: A => B) = xs.map(f)
    def plus(y: List[A]): List[A] = xs ::: y
  }
}

val allTriples = triple() // List((4,3,5), (3,4,5))
```

succeed = pure*: Der Wert gehört zur Lösungsmenge*

fail *: Die Lösungsmenge ist leer*

plus *vereinigt zwei potentielle (!) Lösungsmengen.*

Instanziierung

```
def triple[M[_]: MonadPlus](): M[Triple] =
  for ( i <- MonadPlus[M].choose(2, 3, 4, 5);
        j <- MonadPlus[M].choose(2, 3, 4, 5);
        k <- MonadPlus[M].choose(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k)
               MonadPlus[M].succeed((i, j, k))
             else MonadPlus[M].fail)
    yield r
```

Generischer Algorithmus

Nicht-Determinismus generisch

List als MonadPlus

Instanzierung von MonadPlus mit List

Ok, aber funktioniert das auch mit einem rekursiven solve?

```
def triple[M[_]: MonadPlus](): M[Triple] = {

  def OK(chosen: List[Int]): Boolean = chosen match {
    case i :: j :: k :: _ => i * i + j * j == k * k
  }

  extension (lst: List[Int]) {
    def toTriple: Triple = lst match {
      case i :: j :: k :: _ => (i, j, k)
    }
  }

  def solve(ijk: List[Int]): M[Triple] =
    if  (ijk.length < 3) {
      for (x <- MonadPlus[M].choose(2,3,4,5);
           s <- solve(ijk ++ List(x)))
        yield s
    } else
      if (OK(ijk))
        MonadPlus[M].succeed(ijk.toTriple)
      else MonadPlus[M].fail

  solve(Nil)
}
```

Generischer Algorithmus

```
given MonadPlus[List] with {
  def pure[A](x: A): List[A] = List(x)
  def fail[A]: List[A] = Nil
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] =
      xs.flatMap(f)
    override def map(f: A => B) = xs.map(f)
    def plus(y: List[A]): List[A] = xs ::: y
  }
}
```

Instanz der Typklasse

```
val allTriples = triple()
  List((4,3,5), (3,4,5))
```

Nicht-Determinismus generisch

List als MonadPlus

Instanzierung von MonadPlus mit List

Ok, aber funktioniert das auch mit den n Damen?

```
def queens[M[_]:MonadPlus](n: Int): M[List[Int]] = {

  def Ok(board: List[Int]): Boolean =
    (for (i <- 0 until board.length;
          j <- i + 1 until board.length
          ) yield {
      val (x, y) = (board(i), board(j))
      val d = j - i
      !(x == y || y == x - d || y == x + d)
    }).find(_ == false)
      .getOrElse(true)

  val alternatives = (0 until n).map(List(_)).toList

  def solve(chosen: List[Int]): M[List[Int]] =
    if (Ok(chosen)) {
      if (chosen.length == n) {
        MonadPlus[M].pure(chosen)
      } else {
        for (i: List[Int] <- MonadPlus[M].choose(alternatives);
             s: List[Int] <- solve(chosen ::: i))
          yield s
      }
    } else MonadPlus[M].fail

  solve(Nil)
}
```

Generischer Algorithmus

```
given MonadPlus[List] with {
  def pure[A](x: A): List[A] = List(x)
  def fail[A]: List[A] = Nil
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] =
      xs.flatMap(f)
    override def map(f: A => B) = xs.map(f)
    def plus(y: List[A]): List[A] = xs ::: y
  }
}
```

Instanz der Typklasse

```
val fourQueens = queens(4)
```

```
List(List(1, 3, 0, 2), List(2, 0, 3, 1))
```

Nicht-Determinismus generisch

Backtracking (funktional) als MonadPlus

Instanzierung von MonadPlus mit Backtracking-Implementierung

Dazu brauchen wir einen Backtracking-Typ der Instanz von MonadPlus sein kann

Rückblick: Funktionales Backtracking: BT mit Failure/Sucess – Continuations (siehe Foliensatz 9)

Als Instanz von MonadPlus : ???

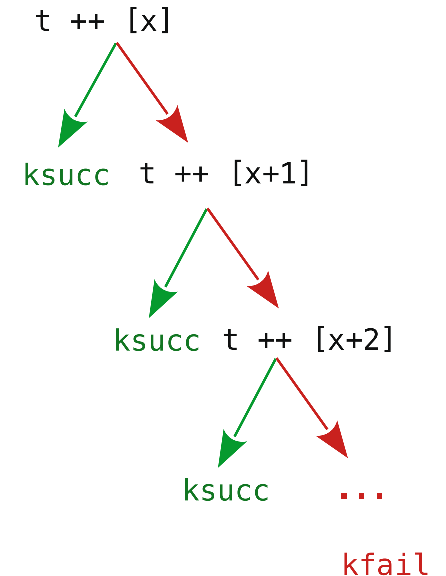
```
def NQueens(n: Int): Unit = {

   def solve(t: List[Int], ksucc: List[Int] => Unit, kfail: => Unit): Unit = {

     def loop(x: Int): Unit = {
       if (x == n)
         kfail // Backtrack
       else {
         val t_extended = t ++ List(x)
         if (Ok(t_extended)) {
           solve(t_extended, ksucc, loop(x + 1))
         } else {
           loop(x + 1)
         }
       }
     }

     if (t.length == n)
       ksucc(t)
     else
       loop(0) // ~ for (x <- 0 until n)
   }

   solve(Nil, lst => println(lst),  println("Failed"))
 }
```



Backtracking (BT) – funktional: Pythagoreische Tripel

Ein pythagoreisches Tripel – BT-Algorithmus funktional

Wir starten wieder mit dem einfacheren Beispiel der pythagoreischen Tripel

Ein funktionaler BT-Algorithmus ist:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit

def triple(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(ijk: List[Int], ksucc: SuccessCont, kfail: FailureCont): Unit = {
    def loop(x: Int) : Unit =
      if (x > 5)
        kfail()
      else {
        val chosen_extended = ijk.appended(x)
        solve(chosen_extended, ksucc, () => loop(x+1) )
      }

    if (ijk.length == 3) {
      if (OK(ijk)) ksucc(ijk.toTriple)
      else kfail()
    } else {
      loop(1)
    }
  }
  solve(Nil, ksucc, kfail)
}
```

mit:

```
type Triple = (Int, Int, Int)

def OK(chosen: List[Int]): Boolean = chosen match {
  case i :: j :: k :: _ => i * i + j * j == k * k
}

extension (lst: List[Int]) {
  def toTriple: Triple = lst match {
    case i :: j :: k :: _ => (i, j, k)
  }
}
```

Backtracking – funktional: Pythagoreische Tripel

Version mit entfalteter Rekursion / so geht es auch:

```
def triple(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(ijk: List[Int], ksucc: SuccessCont, kfail: FailureCont): Unit =
    if (ijk.length == 3) {
      if (OK(ijk)) ksucc(ijk.toTriple)
      else kfail()
    } else {
      solve(
        ijk.appended(1),
        ksucc,
        () => solve(
          ijk.appended(2),
          ksucc,
          () => solve(
            ijk.appended(3),
            ksucc,
            () => solve(
              ijk.appended(4),
              ksucc,
              () => solve(
                ijk.appended(5),
                ksucc, kfail
              )
            )
          )
        )
      )
    }
  solve(Nil, ksucc, kfail)
}
```

Backtracking – funktional: Pythagoreische Tripel

Die beiden Continuations können in eine Klasse / ein Objekt gepackt werden:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit
case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)

def triple(contduo: ContDuo): Unit = {

  def solve(ijk: List[Int], kduo: ContDuo): Unit = {
    def loop(x: Int) : Unit =
      if (x > 5)
        kduo.kfail()
      else {
        val chosen_extended = ijk.appended(x)
        solve(chosen_extended, ContDuo(kduo.ksucc, () => loop(x+1)) )
      }

    if (ijk.length == 3) {
      if (OK(ijk)) kduo.ksucc(ijk.toTriple)
      else kduo.kfail()
    } else {
      loop(1)
    }
  }

  solve(Nil, contduo)
}


triple(ContDuo(triple => println(triple), () => println("Failure")))
```

Backtracking – funktional: Pythagoreische Tripel

Etwas Curry hilft immer:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit

case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)


def triple_CPS: ContDuo => Unit = {

  def solve(ijk: List[Int]) : ContDuo => Unit = {
    def loop(x: Int): ContDuo => Unit =
      if (x > 5)
        kduo => kduo.kfail()
      else
        kduo => solve(ijk.appended(x))(ContDuo(kduo.ksucc, () => loop(x+1)(kduo)) )

    if (ijk.length == 3) {
      if (OK(ijk)) kduo => kduo.ksucc(ijk.toTriple)
      else kduo => kduo.kfail()
    } else {
      kduo => loop(1)(kduo)
    }
  }

  contDuo => solve(Nil)(contDuo)
}


triple(ContDuo(triple => println(triple), () => println("Failure")))
```

Backtracking – funktional: Pythagoreische Tripel

Noch eine Typdefinition: ContDuoToUnit = (ContDuo => Unit)

Das ändert erst mal nicht viel

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit
case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)

type ContDuoToUnit = ContDuo => Unit

def triple_CPS: ContDuoToUnit = {

  def solve(ijk: List[Int]) : ContDuoToUnit = {
    def loop(x: Int): ContDuoToUnit =
      if (x > 5)
        kduo => kduo.kfail()
      else
        kduo => solve(ijk.appended(x))(ContDuo(kduo.ksucc, () => loop(x+1)(kduo)) )

    if (ijk.length == 3) {
      if (OK(ijk)) kduo => kduo.ksucc(ijk.toTriple)
      else kduo => kduo.kfail()
    } else {
      kduo => loop(1)(kduo)
    }
  }

  contDuo => solve(Nil)(contDuo)
}


triple(ContDuo(triple => println(triple), () => println("Failure")))
```

Backtracking – Monadisch

BT – eine (zukünftige) Instanz von MonadPlus

BT als Umschlagklasse für ContDuoToUnit

Beispiel Tripel mit den (noch zu implementierenden) MonadPlus-Methoden

```
type SuccessCont[T] = T => Unit
type FailureCont = () => Unit

case class BT[A](kduo: (ksucc: SuccessCont[A], kfail: FailureCont) => Unit)

extension[A, B] (x: BT[A]) {
  def flatMap(f: A => BT[B]): BT[B] = ???
  def map(f: A => B): BT[B] = ???
  def plus(y: BT[A]): BT[A] = ???
}

def chooseBT[A](alternatives: List[A]): BT[A] = ???
def chooseBT[A](alternatives: A*): BT[A] = chooseBT(alternatives.toList)
def pureBT[A](a: A): BT[A] = ???
def succeedBT[A](a: A): BT[A] = pureBT(a)
def failBT[A]: BT[A] = ???

type Triple = (Int, Int, Int)

def triple(): BT[Triple] =
  for ( i <- chooseBT(2, 3, 4, 5);
        j <- chooseBT(2, 3, 4, 5);
        k <- chooseBT(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k) succeedBT((i, j, k)) else failBT[Triple] )
    yield r
```

BT als Umschlagklasse für ContDuo => Unit

BT – ein monadischer Backtrack-Typ

Triple mit rekursivem Algorithmus

```
def triple(): BT[Triple] = {

  def OK(chosen: List[Int]): Boolean = chosen match {
    case i :: j :: k :: _ => i * i + j * j == k * k
  }
  extension (lst: List[Int]) {
    def toTriple: Triple = lst match {
      case i :: j :: k :: _ => (i, j, k)
    }
  }

  def solve(ijk: List[Int]): BT[Triple] =
    if  (ijk.length < 3) {
      for (x <- chooseBT(2,3,4,5);
           s <- solve(ijk ++ List(x)))
        yield s
    } else {
      if (OK(ijk)) succeedBT(ijk.toTriple)
      else failBT[Triple]
    }

  solve(Nil)
}
```

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ

pure und fail

pure repräsentiert einen Wert, der an die Success-Continuation übergeben werden kann

fail aktiviert die Failure-Continuation

```
def pureBT[A](a: A): BT[A] =
  BT[A]( (ksucc, kfail) => ksucc(a) )


def failBT[A]: BT[A] =
  BT[A]( (ksucc, kfail) => kfail() )
```

BT – ein monadischer Backtrack-Typ

map und flatMap

map delegieren wir der Einfachheit halber an flatMap

flatMap verkettet Berechnungen / setzt eine erfolgreiche Berechnung fort

```
extension[A, B] (x: BT[A]) {

  def flatMap(f: A => BT[B]): BT[B] =
    BT(
      (ks: SuccessCont[B], kf:FailureCont) =>
        x.kduo(
          (a:A) => f(a).kduo(ks, kf),
          kf)
    )

  def map(f: A => B): BT[B] =
    flatMap((a: A) => pureBT(f(a)))

  def plus(y: BT[A]): BT[A] = ???

}
```

f wird auf a angewendet und dann geht es weiter mit ks und kf

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ

choose

wird wieder auf plus reduziert:

```
def chooseBT[A](alternatives: List[A]): BT[A] =
  alternatives.foldLeft(failBT[A])(
    (acc, i) => acc.plus(pureBT(i))
  )

def chooseBT[A](alternatives: A*): BT[A] = chooseBT(alternatives.toList)
```

BT – ein monadischer Backtrack-Typ

plus

plus repräsentiert alternative Ausführungen, wenn die erste fehlschlägt, nimm die zweite

```
extension[A, B] (x: BT[A]) {

  ...

  def plus(y: BT[A]): BT[A] =
    BT[A](
      (ks: SuccessCont[A], kf:FailureCont) =>
        x.kduo(
          a => ks(a),
          () => y.kduo(ks, kf)
        )
    )
}
```

Probiere es mit der einen Berechnung, x. Wenn das schief geht, dann probiere es mit der anderen, y.

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ

Hmm – So einfach nicht!

```
tripleA().kduo( (res: Triple) => println(res), () => println("failure"))
```

~> failure

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ / verbessert

plus und flatMap

Die Definitionen sind Typ-korrekt und für sich überzeugend,
aber flatMap (Verkettung von Aktionen) und plus (Auswahl von Aktionen) kooperieren nicht.

flatMap muss „von links über plus distribuieren“:

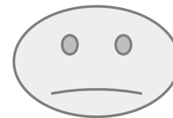
(x plus y) flatMap f = (x flatMap f) plus (y flatMap f)

es muss also egal sein, ob ich

- **eine Wahl zwischen x und y treffe und dann mit f weiter mache, oder ob ich**
- **Eine Wahl treffe zwischen „x und dann f“ oder „y und dann f“**

Dieses „von links distribuieren“ ist bei den Definitionen, so wie sie sind, nicht gewährleistet.

```
def flatMap(f: A => BT[B]): BT[B] =
  BT(
    (ks: SuccessCont[B], kf:FailureCont) =>
      x.kduo(
        (a:A) => f(a).kduo(ks, kf),
        kf)
  )
```



f „kommt nicht in die failure-Continuation“!

BT – ein monadischer Backtrack-Typ / verbessert

plus und flatMap

Definiere plus und flatmap derart, dass flatMap von links über plus distribuiert:

(x plus y) flatMap f = (x flatMap f) plus (y flatMap f)

Hier einfach mit einer angepassten Klasse, die Plus erhält:

```
enum BT[A] {
  case Plus(x: BT[A], y: BT[A])
  case Cont(k2U: (SuccessCont[A], FailureCont) => Unit )
  def apply(ks: SuccessCont[A], kf: FailureCont): Unit = this match {
    case Cont(k2U) => k2U(ks, kf)
    case Plus(x, y) =>
      ( (ks: SuccessCont[A], kf: FailureCont) =>
          x.apply( a => ks(a), () => y(ks, kf) )
      ).apply(ks, kf)
  }
}
import BT._
```

Die Plus-Variante speichert die Alternativen.

Die Cont-Variante entspricht der bisherigen BT-Definition.

Bei der Anwendung / Ausführung kann in der Plus-Variante auf die Alternativen x und y zugegriffen werden und y wird zur Failure-Continuation von x.

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ / verbessert

plus und flatMap

Definiere plus und flatmap derart, dass flatMap von links über plus distribuiert:

(x plus y) flatMap z = (x flatMap z) plus (y flatMap z)

```
extension[A, B] (x: BT[A]) {
  def flatMap(f: A => BT[B]): BT[B] = x match {
    case Cont(k2U) =>
      Cont(
        (ks, kf) =>
          k2U((a:A) =>
            f(a) match {
              case Cont(k2U) => k2U(ks, kf)
              case Plus(x, y) =>
                x(
                  a => ks(a),
                  () => y(ks, kf)
                )
            }, kf)
      )

    case Plus(x, y) =>
      Plus[B](x.flatMap(f), y.flatMap(f))
  }

  def map(f: A => B): BT[B] = flatMap((a: A) => pureBT(f(a)))

  def plus(y: BT[A]): BT[A] = Plus(x, y)
}
```

Cont-Variante: wie oben

Plus-Variante: x und falls das schief geht y

flatMap wird von links über Plus distributiert

plus speichert die Alternativen in einem Plus

Backtracking – Monadisch

BT – ein monadischer Backtrack-Typ / verbessert

choose / pure / succed / fail

da ändert sich nichts:

```
def chooseBT[A](alternatives: List[A]): BT[A] =
  alternatives.foldLeft(failBT[A])(
    (acc, i) => acc.plus(pureBT(i))
  )

def chooseBT[A](alternatives: A*): BT[A] = chooseBT(alternatives.toList)

def pureBT[A](a: A): BT[A] =
  Cont[A]( (ksucc, kfail) => ksucc(a) )

def failBT[A]: BT[A] =
  Cont[A]( (ksucc, kfail) => kfail() )

def succeedBT[A](a: A): BT[A] = pureBT(a)
```

BT – ein monadischer Backtrack-Typ / verbessert

Backtracking mit BT:

```
type Triple = (Int, Int, Int)

def tripleA(): BT[Triple] =
  for ( i <- chooseBT(2, 3, 4, 5);
        j <- chooseBT(2, 3, 4, 5);
        k <- chooseBT(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k)
               succeedBT((i, j, k))
             else failBT[Triple] )
    yield r
```

Mit for-Comprehension

```
def tripleB(): BT[Triple] = {

  def OK(chosen: List[Int]): Boolean = chosen match {
    case i :: j :: k :: _ => i * i + j * j == k * k
  }
  extension (lst: List[Int]) {
    def toTriple: Triple = lst match {
      case i :: j :: k :: _ => (i, j, k)
    }
  }

  def solve(ijk: List[Int]): BT[Triple] =
    if  (ijk.length < 3) {
      for (x <- chooseBT(2,3,4,5);
           s <- solve(ijk ++ List(x)))
        yield s
    } else {
      if (OK(ijk)) succeedBT(ijk.toTriple)
      else failBT[Triple]
    }

  solve(Nil)
}
```

Mit Rekursion

```
tripleA()( (res: Triple) => println(res), () => println("failure"))
tripleB()( (res: Triple) => println(res), () => println("failure"))
```

Backtracking – Monadisch

BT – eine Instanz von MonadPlus

Jetzt bietet BT alles um ein MonadPlus zu sein

Zur Vermeidung von Konfusion (des Compilers ?) werden die BT-Methoden mit BT markiert:

```
enum BT[A] {
  case Plus(x: BT[A], y: BT[A])

  case Cont(k2U: (SuccessCont[A], FailureCont) => Unit )

  def apply(ks: SuccessCont[A], kf: FailureCont): Unit =
    this match {
      case Cont(k2U) => k2U(ks, kf)
      case Plus(x, y) =>
        ( (ks: SuccessCont[A], kf: FailureCont) =>
            x.apply( a => ks(a), () => y(ks, kf) )
        ).apply(ks, kf)
    }
}
import BT._
```

```
extension[A, B] (x: BT[A]) {
  def flatMapBT(f: A => BT[B]): BT[B] = x match {
    case Cont(k2U) =>
      Cont(
        (ks, kf) =>
          k2U((a:A) =>
            f(a) match {
              case Cont(k2U) => k2U(ks, kf)
              case Plus(x, y) =>
                x(
                  a => ks(a),
                  () => y(ks, kf)
                )
            }, kf)
      )
    case Plus(x, y) =>
      Plus[B](x.flatMapBT(f), y.flatMapBT(f))
  }

  def mapBT(f: A => B): BT[B] =
    flatMapBT((a: A) => pureBT(f(a)))

  def plusBT(y: BT[A]): BT[A] =
    Plus(x, y)
}
```

BT mit der notwendigen MonadPlus Funktionalität ausstatten

Backtracking – Monadisch

BT – eine Instanz von MonadPlus

Jetzt bietet BT alles um ein MonadPlus zu sein
und kann als MonadPlus-Instanz definiert werden:

```
given MonadPlus[BT] with {
  def pure[A](x: A): BT[A] = pureBT(x)
  def fail[A]: BT[A] = failBT[A]
  extension [A, B](x: BT[A]) {
    def flatMap(f: A => BT[B]): BT[B] = x.flatMapBT(f)
    override def map(f: A => B) = x.mapBT(f)
    def plus(y: BT[A]): BT[A] = x plusBT (y)
  }
}
```

BT als Instanz von MonadPlus erklären

BT – eine Instanz von MonadPlus

Generische Backtracking Algorithmen können ausgeführt werden / Beispiel 1

```
def tripleGen[M[_]: MonadPlus]: M[Triple] =
  for ( i <- MonadPlus[M].choose(2, 3, 4, 5);
        j <- MonadPlus[M].choose(2, 3, 4, 5);
        k <- MonadPlus[M].choose(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k)
          MonadPlus[M].succeed((i, j, k))
        else MonadPlus[M].fail)
    yield r

def t[M[_]: MonadPlus]: M[Triple] = tripleGen


t.apply( (res: Triple) => println(res), () => println("failure"))
```

BT – eine Instanz von MonadPlus

Generische Backtracking Algorithmen können ausgeführt werden / Beispiel 2

```scala
def queensGen[M[_]:MonadPlus](n: Int): M[List[Int]] = {

  def Ok(board: List[Int]): Boolean =
    (for (i <- 0 until board.length;
          j <- i + 1 until board.length
          ) yield {
      val (x, y) = (board(i), board(j))
      val d = j - i
      !(x == y || y == x - d || y == x + d)
    }).find(_ == false)
      .getOrElse(true)

  val alternatives = (0 until n).map(List(_)).toList

  def solve(chosen: List[Int]): M[List[Int]] =
    if (Ok(chosen)) {
      if (chosen.length == n) {
        MonadPlus[M].pure(chosen)
      } else {
        for (i: List[Int] <- MonadPlus[M].choose(alternatives);
             s: List[Int] <- solve(chosen ::: i))
          yield s
      }
    } else MonadPlus[M].fail

  solve(Nil)
}

def q[M[_]: MonadPlus]: M[List[Int]] = queensGen(4)

q.apply( (res: List[Int]) => println(res), () => println("failure"))
```