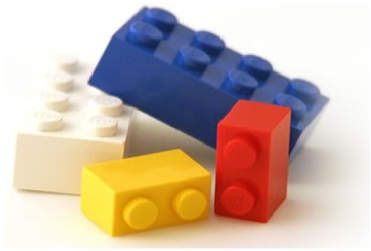
Software-Komponenten

Th. Letschert

THM

University of Applied Sciences



Fortsetzungs-Monaden

- CPS: Continuation Passing Style
- Continuation-Monade

CPS – Continuation Passing Style

Fortsetzungs-Funktion (Continuation / Callback) als Parameter

CPS-Stil: Eine Funktion hat den Verwender ihres Ergebnisses als Parameter

Beispiel 1:

```
def fac(x: Int) : Int =
  if (x == 0) 1
  else fac(x-1)*x
```

„Normale Funktion“: Liefert Ergebnis

```
def facC(x: Int, cont: Int => Unit) : Unit =
  if (x == 0) cont(1)
  else facC(x-1,
    res => cont(res*x))
```

Funktion im „CPS-Stil“: Hat einen zusätzlichen Funktionsparameter (continuation, oder callback genannt), der das Ergebnis annimmt.

```
val res_direct = fac(10)
// res_direct = 3628800

var res_cont = 0

facC(10, res => {res_cont = res} )
// res_cont = 3628800
```

CPS – Continuation Passing Style

Fortsetzungs-Funktion (Continuation / Callback) als Parameter

Beispiel 2:

```
def mult2(x: Int): Int = x*2
def add1(x: Int): Int = x+1

val output_direct =
  println(
    mult2(
      add1(
        20)))
```

„Normale“ Funktionen: Liefern Ergebnis

Parameterübergaben: Verkettung von Funktionen im „normalen“ Stil.

```
def pure(x: Int, k: Int => Unit): Unit = k(x)
def mult2C(x: Int, k: Int => Unit): Unit = k(x*2)
def add1C(x: Int, k: Int => Unit): Unit = k(x+1)

val output_cont =
  pure(20, x =>
    add1C(x, y =>
      mult2C(y, z =>
        println(z))))
```

Funktionen im CPS-Stil: Haben einen zusätzlichen Funktionsparameter (continuation, oder callback genannt), der das Ergebnis annimmt.

Verkettung von Funktionen im CPS-Stil.

CPS – Continuation Passing Style

Asynchrone Verarbeitung

CPS und Asynchronität

Manche Plattformen ohne „richtige Concurrency“ haben vordefinierte (I/O-) Funktionen, die grundsätzlich asynchron abgewickelt werden und an die weitere Aktionen als Callbacks angehängt werden können. Diese werden dann ebenfalls asynchron abgewickelt.

Beispiel:

```
fs.readFile(
   '...',
   function (err, data) {
      if (err) throw err;
       ....
});
```

CSP / Callback: Ausführung im Thread von readFile (vordefiniert asynchron)

```
f = Future{ fs.readFile('...') }

f onComplete {
   case Success(..) => ..
   case Failure(..) => ..
}
```

Future / Promise („richtige Concurrency“): Thread kommt aus einem Threeadpool

CPS – Continuation Passing Style

Asynchrone Verarbeitung

CPS und Asynchronität

Future und CPS:

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def pure(x: Int, k: Int => Unit): Unit = Future { k(x) }
def mult2C(x: Int, k: Int => Unit): Unit = Future { k(x*2) }
def add1C(x: Int, k: Int => Unit): Unit = Future { k(x+1) }

def printC(x: Int): Unit = Future{ println(x) }

def run(): Unit =
  pure(20, x =>
    add1C(x, y =>
      mult2C(y, z =>
        printC(z)
      )
    )
  )
```

~

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def pure(x: Int): Future[Int] = Future { x }
def mult2C(x: Int): Future[Int] = Future { x*2 }
def add1C(x: Int): Future[Int] = Future { x+1 }
def printC(x: Int): Future[Unit] = Future{ println(x) }

def run(): Unit =
  pure(20) foreach(x =>
    add1C(x) foreach( y =>
      mult2C(y) foreach( z =>
          printC(z)
        )
      )
  )
```

Asynchrone Verarbeitung

CPS und Asynchronität

Future ist monadisch:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def pure(x: Int): Future[Int] = Future { x }
def mult2C(x: Int): Future[Int] = Future { x*2 }
def add1C(x: Int): Future[Int] = Future { x+1 }
def printC(x: Int): Future[Unit] = Future{ println(x) }
```

```
def run(): Unit =
  pure(20) foreach(x =>
    add1C(x) foreach( y =>
      mult2C(y) foreach( z =>
          printC(z)
        )
      )
  )
```

Unschön, kompliziert

~

```
def run(): Unit =
  for (x <- pure(20);
       y <- mult2C(x);
       z <- add1C(y) )
  yield printC(z)
```

Glücklicherweise ist Future ein monadischer Typ

Tail-Recursion

CPS und Tail-Recursion (End-Rekursion)

End-Rekursion: Der rekursive Aufruf ist die letzte Aktion im rekursiven Zweig

End-Rekursive Funktionen können in Schleifen umgewandelt werden (in funktionalen Sprachen automatisch durch den Compiler)

Beispiel GGT:

```
@tailrec
def gcd(x: Int, y: Int): Int =
  if (x == y) x
  else gcd(Math.max(x,y) - Math.min(x,y), Math.min(x,y))
```

~

```
def gcd(x: Int, y: Int): Int = {
  var (x_, y_) = (x, y)
  while (x_ != y_) {
    val (a, b) = (Math.max(x_, y_) - Math.min(x_, y_), Math.min(x_, y_))
    x_ = a
    y_ = b
  }
  x_
}
```

Tail-Recursion

CPS und Tail-Recursion

Lineare-Rekursion: Der rekursive Zweig enthält nur einen rekursiven Aufruf – das muss aber nicht die letzte Aktion sein

Viele wichtige Funktionen sind linear-rekursiv, nur sehr wenige sind end-rekursiv

Beispiel linear-rekursive Funktionen:

```
def fac(x: Int) : Int =
  if (x == 0) 1
  else fac(x-1) * x
```

```
def append[A, B >: A](lst1: List[A], lst2: List[B]): List[B] = lst1 match {
  case Nil => lst2
  case head :: tail => head :: append(tail, lst2)
}
```

```
def reverse[A](lst: List[A]): List[A] = lst match {
  case Nil => Nil
  case head :: tail => append(reverse(tail), List(head))
}
```

Tail-Recursion

CPS und Tail-Recursion

Beobachtung: Linear-rekursive Funktionen sind end-rekursiv nach Transformation in CPS

```
@tailrec
def facC(x: Int, k: Int => Unit) : Unit =
  if (x == 0) k(1)
  else facC(x-1, i => k(i*x))

@tailrec
def appendC[A, B >: A](lst1: List[A], lst2: List[B],
                       k: List[B] => Unit): Unit = lst1 match {
  case Nil => k(lst2)
  case head :: tail => appendC(tail, lst2, l => k(head::l))
}

@tailrec
def reverseC[A](lst: List[A],
                k: List[A] => Unit): Unit = lst match {
  case Nil => k(Nil)
  case head :: tail =>
    reverseC(tail, l => appendC(l, List(head), k))
}

def pure[A](a: A, k: A => Unit): Unit = k(a)

val lst_1 = List(1, 2, 3)
val lst_2 = List(4, 5, 6)
```

```
def print42: Unit =
  pure(lst_1, x =>
    appendC(x, lst_2, y =>
      reverseC(y, z =>
        println(z) // List(6, 5, 4, 3, 2, 1)
      )
    )
  )
```

Backtracking

CPS und „funktionale Emulation“ imperativer Kontrollstrukturen Beispiel Backtracking

N-Damen / imperativ mit Exceptions

```
def NQueens(n: Int): List[Int] = {

  class BTException extends Throwable

  def solve(t: List[Int]): List[Int] = {
    if (t.length == n)
      return t
    else {
      for (x <- 0 until n) {
        val t_extended = t ++ List(x)
        if (Ok(t_extended))
          try {
            return solve(t_extended)
          } catch { // gesicherten Stand festhalten
            case _: BTException =>
          }
      }
      throw new BTException // Sprung (Backtrack) zum letzten gesicherten Stand
    }
  }

  try {
    solve(Nil)
  } catch {
    case e:BTException => Nil
  }
}

println(NQueens(4)) // List(1, 3, 0, 2)
```

```
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)
```

Backtracking

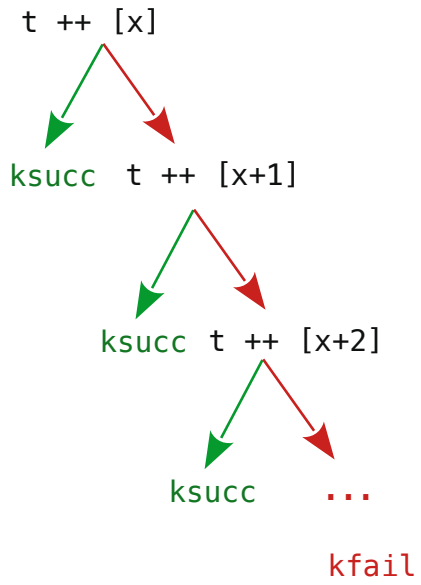
CPS und „funktionale Emulation“ imperativer Kontrollstrukturen Beispiel Backtracking

N-Damen / mit funktionaler Emulation der Exceptions

Der Kontrollfluss wird durch 2 Continuations, `ksucc` und `kfail`, modelliert und manipuliert

```
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"))
 }

}
```

Versuche es mit t ++ [x], wenn das schief geht, dann mache weiter mit der nächsten Runde (x+1) in der Schleife.




```
NQueens(4) // List(1, 3, 0, 2)
```

Continuation-Monade

Vereinfachten Verwendung des CPS

Das Problem: CPS ~> Verknüpfung als komplizierte Verschachtelung

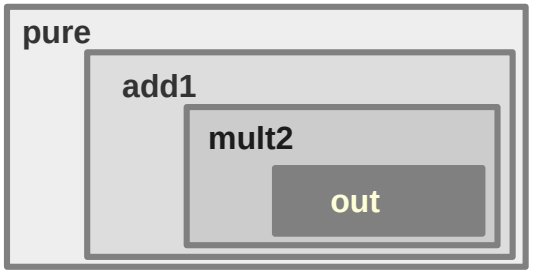
Beispiel:

```
def pure(x: Int, k: Int => Unit): Unit = k(x)
def mult2(x: Int, k: Int => Unit): Unit = k(x*2)
def add1(x: Int, k: Int => Unit): Unit = k(x+1)
def out(x: Int): Unit = println(x)
```

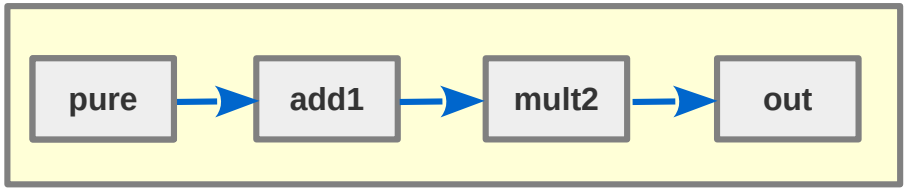
```
print42 ~ println( (20 + 1) * 2 )
```

```
def print42 : Unit =
  pure(20, x =>
    add1(x, y =>
      mult2(y, out)
    )
  )
```

Unübersichtliche Verschachtelung: Folgeoperationen müssen in bisherige Strukturen eingebaut werden.



print42



print42

Den Code hätten wir aber lieber in dieser „kombinatorischen Form“ als Verkettung, statt als Verschachtelung von Funktionen.

Continuation-Monade

Currying: Etwas Curry-Pulver hilft immer

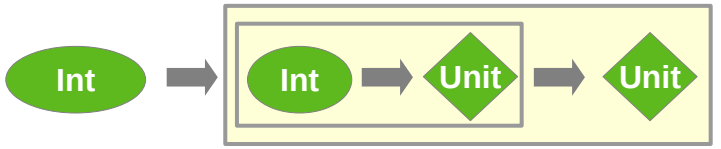
```
def pure(x: Int):  (Int => Unit) => Unit =
  kInt => kInt(x)

def mult2(x: Int): (Int => Unit) => Unit =
  kInt => kInt(x*2)

def add1(x: Int):  (Int => Unit) => Unit =
  kInt => kInt(x+1)

def out: Int => Unit = x => println(x)

def print42 : Unit =
  pure(20)( x =>
    add1(x)( y =>
      mult2(y)(out)
    )
  )
```

Continuation-Monade

Beobachtung: flatMap-Signatur

Die zu verkettenden Funktionen haben eine „flatMap-kompatible“ Signatur wenn `(Int => Unit) => Unit` ein Funktor ist

```
def pure:  Int => (Int => Unit) => Unit

def mult2: Int => (Int => Unit) => Unit

def add1:  Int => (Int => Unit) => Unit
```



Funktor F[•] mit

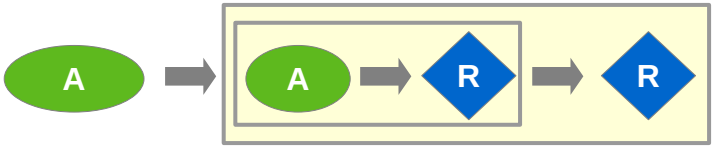
F[Int] ~ `(Int => Unit) => Unit`

Continuation-Monade

Verallgemeinerung der Ergebnis-Typs

Der Typ der Endergebnisses R wird fixiert: Funktor F[•] = $Cont_{Unit}$ [Int]

```
type Cont[R, A] = (A => R) => R      // Continuation-Monade
type ContToUnit[A] = Cont[Unit, A]   // bei fixiertem (Return Type) R
```

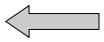


Funktor $Cont_R$ [A]

Continuation-Monade

Monadische Umschlag-Klasse für `(A => R) => R`

```
case class Cont[R, A](kToR: (A => R) => R){
  def map[B](f: A => B): Cont[R, B] = ???
  def flatMap[B](f: A => Cont[R, B]) = ???
}
```



```
type Cont[R, A] = (A => R) => R
```

In einem Umschlag

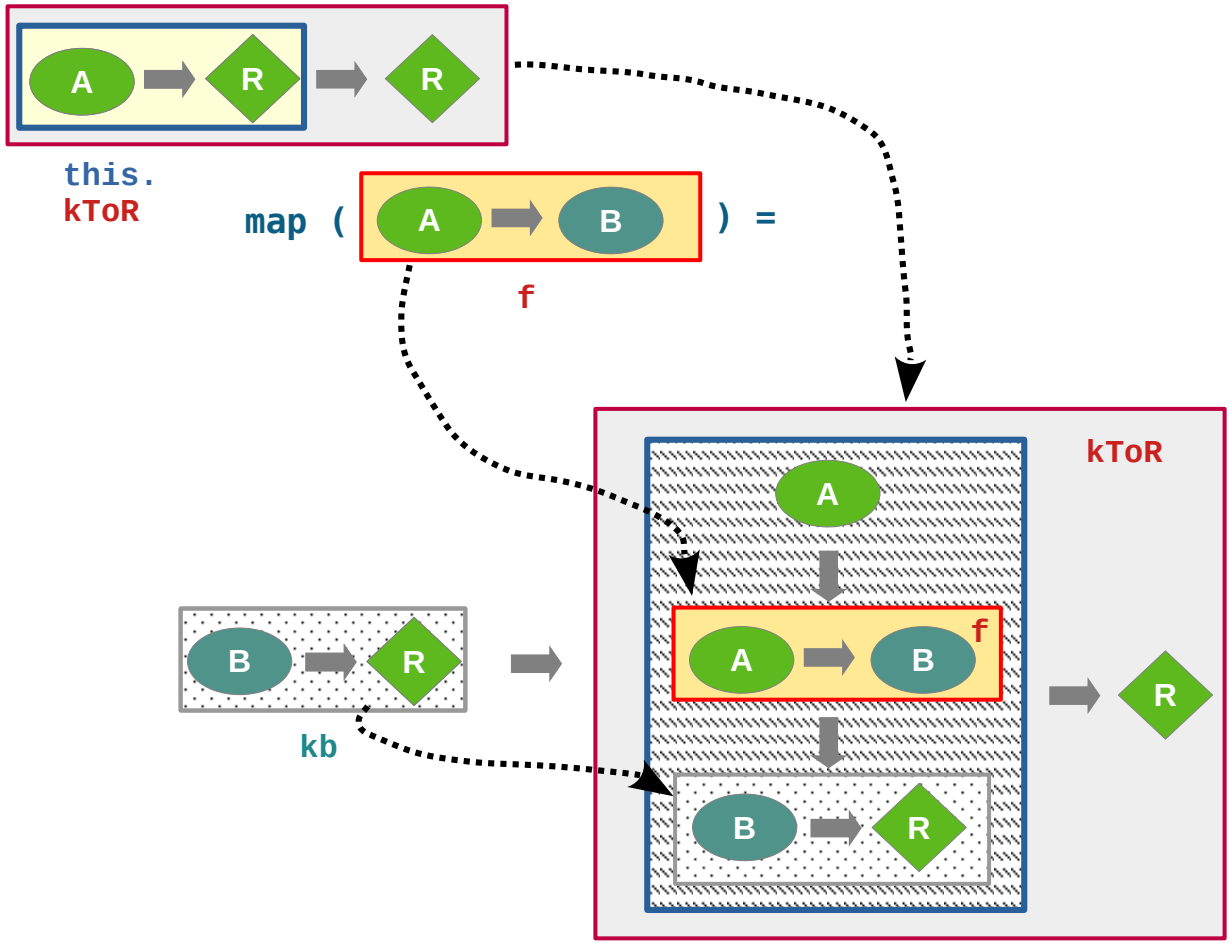
```
def pure[A,R]:  A => Cont[R, A] =
  a => Cont(kA => kA(a))

type ContToUnit[A] = Cont[Unit, A]

def mult2(x: Int): ContToUnit[Int] =
  Cont(kInt => kInt(x*2))

def add1(x: Int): ContToUnit[Int] =
  Cont(kInt => kInt(x+1))

def compute42 : ContToUnit[Int] =
  for (x <- pure(20);
       y <- add1(x);
       z <- mult2(y))
    yield z

def print42 = compute42(x => println(x))
```

Die Typen passen. Das ist schon mal ein Anfang.

Continuation-Monade

Typ als Klasse mit map

```
case class Cont[R, A](kToR: (A => R) => R) {

  def map[B](f: A => B): Cont[R, B] =
    Cont( (kb: B => R) => kToR( (a:A) => kb( f(a)) ))
```



Map: Konstruiere etwas, das mit einer Fortsetzungsfunktion

kb: B => R

zu einem Ende mit R führt.

Nutze dazu ein

- **f: A => B** *und ein*
- **kToR: (A => R) => R**

Lösung: Füttere

- **kToR** *mit*
- **{ a => kb(f(a)) }**

Continuation-Monade

Typ als Klasse mit map

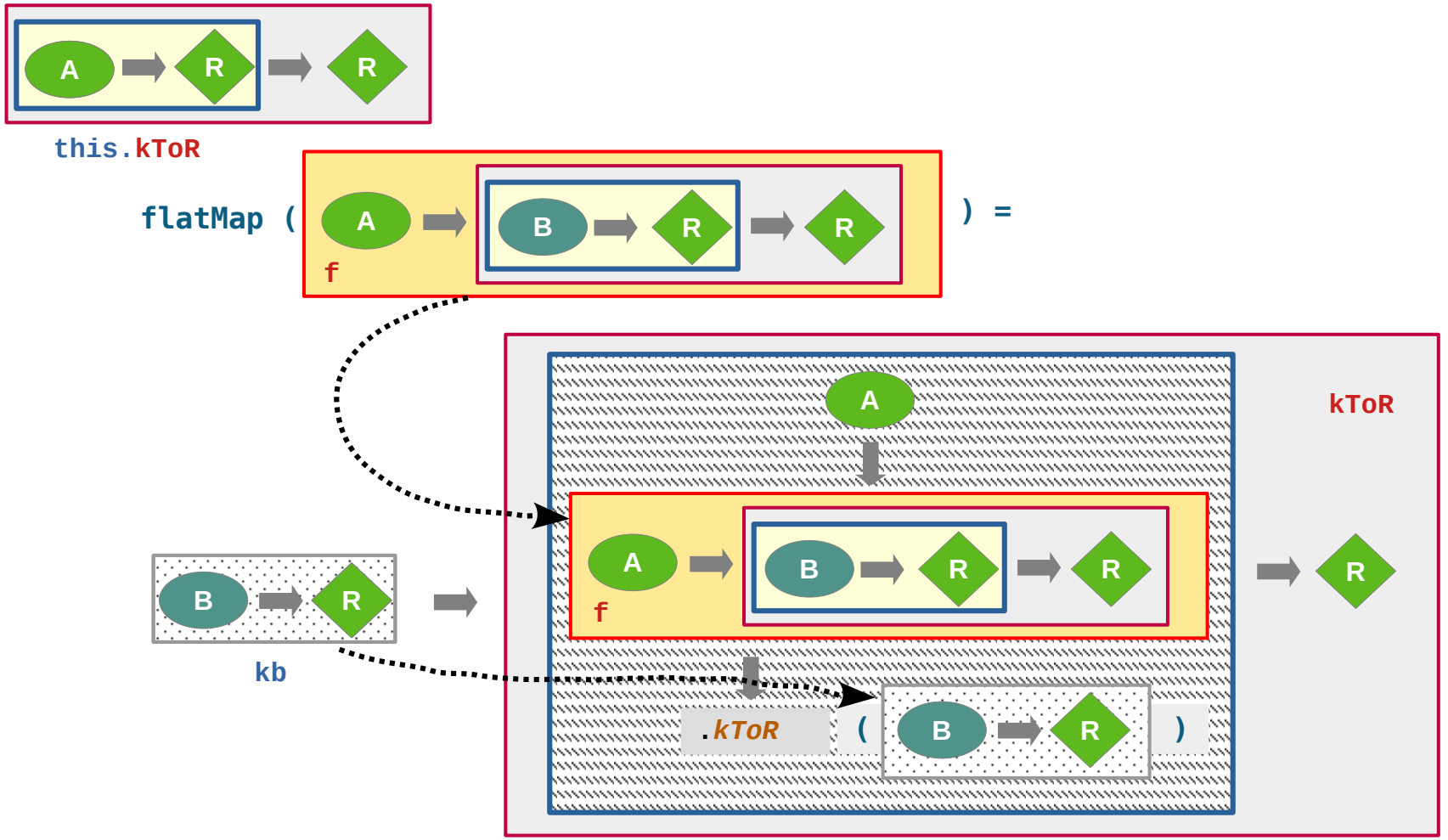
```
class Cont[R, A](kToR: (A => R) => R){
  def map[B](f: A => B): Cont[R, B] =
    Cont( (kb: B => R) => kToR( (a:A) => kb( f(a)) ))
  def flatMap[B](f: A => Cont[R, B]) = ???
}
```

✓

Continuation-Monade

Typ als Klasse mit flatMap

```
case class Cont[R, A](kToR: (A => R) => R) {
  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont( (kb: B => R) => kToR(a => f(a).kToR(kb)) )
```



Continuation-Monade

Typ als Klasse mit map, flatMap, pure und apply

pure und apply zur bequemeren Verwendung

```
case class Cont[R, A](kToR: (A => R) => R){
  def apply(ka: A => R): R = kToR(ka)

  def map[B](f: A => B): Cont[R, B] =
    Cont( (kb: B => R) => kToR( (a:A) => kb( f(a)) ))
  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont( (kb: B => R) => kToR(a => f(a)(kb)) )
}

def pure[A,R]: A => Cont[R, A] =
  a => Cont(kA => kA(a))
```

Continuation-Monade

map via flatMap

Nach den Monaden-Gesetzen muss gelten

```
def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
  Cont( (kb: B => R) => kaToR(a => f(a)(kb)) )

def pure[A,R]:  A => Cont[R, A] =
  a => Cont(kA => kA(a))
```

```
def map[B](f: A => B): Cont[R, B] = flatMap(a => pure(f(a)))
```

Wir haben definiert:

```
def map[B](f: A => B): Cont[R, B] =
  Cont( (kb: B => R) => kaToR( (a:A) => kb( f(a)) ))
```

Es sollte also gelten:

```
flatMap(a => pure(f(a)))   ≈   Cont( (kb: B => R) => kaToR( (a:A) => kb( f(a)) ))
```

Das ist der Fall:

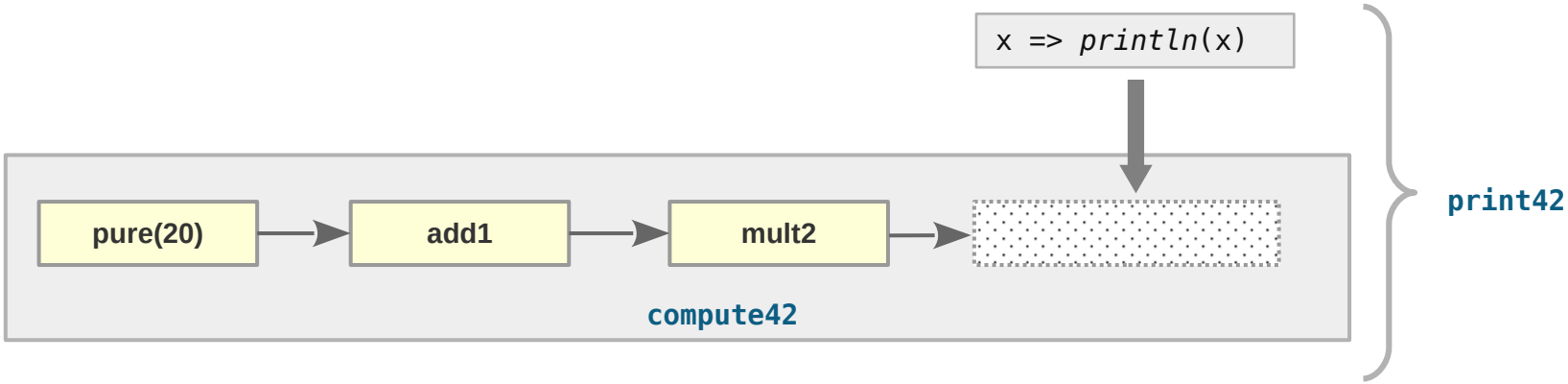
```
flatMap(a => pure(f(a)))
= flatMap(a => Cont(kA => kA(f(a))))
= Cont( (kb: B => R) => kToR(a => (a => Cont(kA => kA(f(a))))(a)(kb)) )
= Cont( (kb: B => R) => kToR(a => (a => kb(f(a))))(a)) )
= Cont( (kb: B => R) => kToR(a => kb(f(a))) )
```

Continuation-Monade

Typ als Klasse mit map, flatMap, pure und apply

im Einsatz

```
type ContToUnit[A] = Cont[Unit, A]

def mult2: Int => ContToUnit[Int] =
  x => Cont(kInt => kInt(x*2))

def add1:  Int => ContToUnit[Int] =
  x => Cont(kInt => kInt(x+1))

def compute42 : ContToUnit[Int] =
  for (x <- pure(20);
       y <- add1(x);
       z <- mult2(y))
    yield z

def print42 = compute42(x => println(x))
```



Continuation-Monade

Beispiel Fakultäten

```
def computeFac: Int => Cont[Unit, Int] = x =>
  if (x == 0)
    pure(1)
  else
    for( y <- computeFac(x-1) )
      yield x * y

def printfac(n: Int) = computeFac(n)(res => println(res))


printfac(10)
```

Continuation-Monade

Continuation-Monade und Future

`Cont[R, A]` repräsentiert / kapselt eine Funktion kToR: `(A => R) => R`

Das Argument von kToR ist der Verarbeiter des von `Cont[R, A]` produzierten Ergebnisses

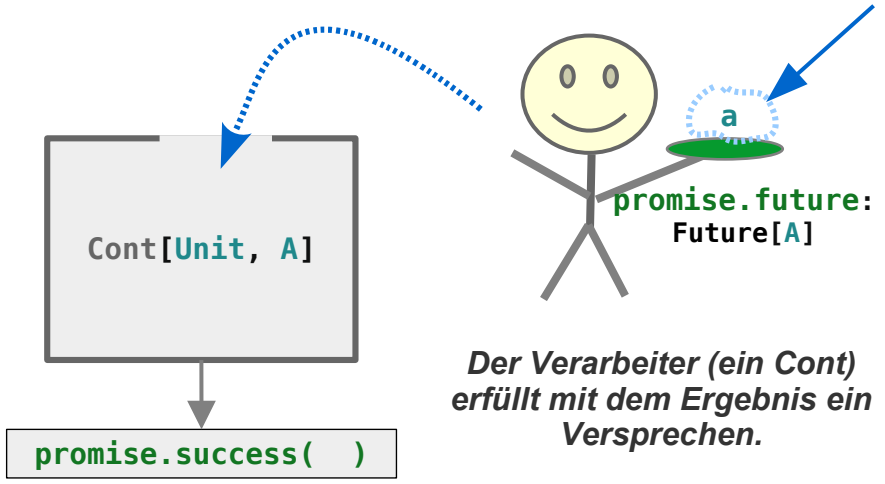
Der Verarbeiter kann jemand sein, der mit dem Ergebnis ein Versprechen (Promise) erfüllt

```
def fac: Int => Cont[Unit, Int] = x =>
  if (x == 0)
    pure(1)
  else
    for( y <- fac(x-1))
      yield x * y
```

```
import scala.concurrent.{Future, Promise, ExecutionContext}
import ExecutionContext.Implicits.global
```

```
// Cont => Future
def futureRes[A](c: Cont[Unit, A]): Future[A] = {
  val promise = Promise[A]
  c(a => promise.success(a))
  promise.future
}
```

```
for (
  x <- futureRes(fac(10))
) yield println(x)

Thread.sleep(1000)
```



Der Verarbeiter (ein Cont) erfüllt mit dem Ergebnis ein Versprechen.