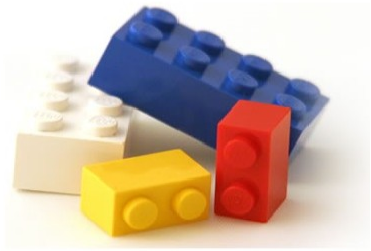
Software-Komponenten

Th. Letschert

THM

University of Applied Sciences



Leser- und Schreiber-Monade

- Leser
- Schreiber

Monade

Metaphern

Metaphern (bildhafte Darstellung) der Monade*

- **Container** *Monaden sind „Burritos“*
- **Berechnungen** *Monaden sind „Berechnungen in einem Kontext“*
  auch „computational effects“

Burrito**

Diese beiden beispielhaften Verwendungen des Konzepts Monade

- **sind nicht immer streng zu trennen**
- **beschreiben unterschiedliche Szenarios der Verwendung des Monaden-Begriffs die inhaltlich oft wenig miteinander zu tun haben**

Monade: Die Strukturierung des Codes erfolgt nach dem gleichen Prinzip,

- **inhaltliche Bezügen können, müssen aber nicht bestehen.**
- **Metaphern sind oft hilfreich, sie sollten aber nicht überschätzt oder als „Realität“ gesehen werden.**

“Monade“ ist nicht mehr als die Strukturierung von Code
nach einem einheitlichen Prinzip mit dem
Generizität und Wiederverwendbakeit erreicht werden.

*Siehe dazu etwa http://tomasp.net/academic/papers/monads/

** Burrito = mexikanischer Börek,
Bildquelle: https://commons.wikimedia.org/wiki/File:Papa_chevos_burrito.jpg

Monade

Monaden-Metapher : „Monaden sind Berechnungen in einem Kontext“

Verallgemeinerung von Container zu **Kontext**

- **M** : *Berechnungen in einem „Kontext“*
- **flatMap** : *Verkettung solcher Berechnungen*

„Kontext“ und „Berechnung in einem Kontext“ kann sehr viel bedeuten

Entscheidend ist, dass die „Berechnung in einem Kontext“ `f` folgenden Typ hat:

```
f: A => M[B]
```

Dabei wird aus einem Wert `a:A` ein Wert `b:B` in einer „Kontext“ `M` berechnet.

```
ma.flatMap(f) = mb: M[B]
```

Der Begriff „Berechnungs-Kontext“ kann auf sehr unterschiedliche Arten interpretiert werden.

z.B. (vorheriger Foliensatz):

- **Listenartige Monaden**: Berechnungen in verschachtelten Iterationen
- **Fehlermanagement-Monaden**: Berechnungen im Kontext eines „Fehler-Notausgangs“

Monaden-Metapher : „Monaden sind Berechnungen in einem Kontext“

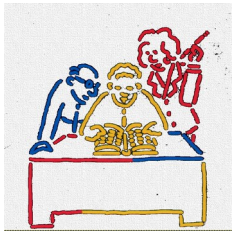
Verallgemeinerung von Container zu Kontext

- **M :** *Berechnungen in einem „Kontext“*
- **flatMap :** *Verkettung solcher Berechnungen*

„Kontext“ und „Berechnung in einem Kontext“ kann sehr viel bedeuten

Zwei weitere Interpretationen von „Kontext“ werden hier behandelt:

- **Reader-Monade:**
  Der Kontext „liefert Werte“ / „man kann ihn lesen“
- **Writer-Monade:**
  Der Kontext „konsumiert Werte“ / „man kann ihn beschreiben“



Reader-Monade

Berechnung in einem Kontext der Werte liefern kann

Bei einer Reader-Monade stellt der Kontext
einige für die Berechnung benötigte Werte zur Verfügung

Mit einer Reader-Monade

- **strukturiert man darum Berechnungen,**
  die von (unveränderlichen) Kontext-Werten abhängen.
- **Sie sind damit ein Idiom / Muster für (funktionale) *Dependency Injection***

Beispiel Login-Daten prüfen

Kontext / Dependency : DB, Informationen über Benutzer

```
trait DB {
  def getId(name: String): Option[Int]
  def getPassword(id: Int): Option[String]
  def getAccessRight(id: Int): Option[AccessRight]
}

enum AccessRight {
    case NoAccess
    case ReadAccess
    case WriteAccess
    case ReadWriteAccess
  }
```

Schnittstelle des Kontexts, von dem die Berechnung abhängt.

Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Version 1a: Explizite Weitergabe des Kontexts via Parameterübergabe

```
def findId(db: DB, name: String): Int =
  db.getId(name).get

def checkPassword(db: DB, id: Int, password: String): Boolean =
  db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }

def getAccessRight(db: DB, id: Int): AccessRight =
  db.getAccessRight(id).get


def login(db: DB, name: String, password: String): AccessRight = {
  val id = findId(db, name)
  val pwOk = checkPassword(db, id, password)
  if (pwOk) getAccessRight(db, id) else AccessRight.NoAccess
}


val accessRight = login(someDB, "Hugo", "56")
```

Einige Routinen, die einen Parameter vom Typ DB benötigen:

Routinen die im Kontext einer DB arbeiten.

Unschön: Das db-Argument muss ständig mitgeschleppt werden.

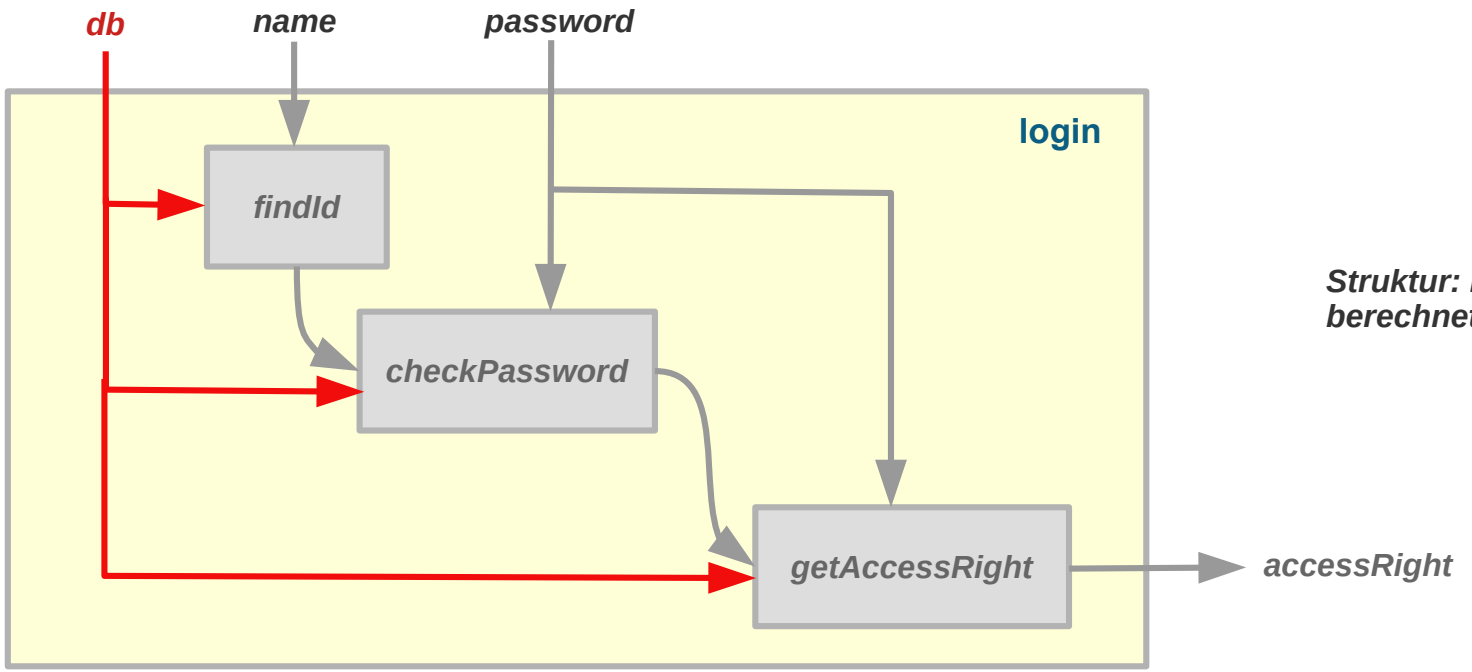
Bemerkung: Zwecks Übersichtlichkeit ist dieser Code unrealistisch einfach. Auf eine Fehlerbehandlung wird beispielsweise weitgehend verzichtet.

Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Version 1a: Explizite Weitergabe des Kontexts via Parameterübergabe

```
def login(db: DB, name: String, password: String): AccessRight = {
  val id = findId(db, name)
  val pwOk = checkPassword(db, id, password)
  if (pwOk) getAccessRight(db, id) else AccessRight.NoAccess
}
```



Struktur: Kombination von berechneten Werten.

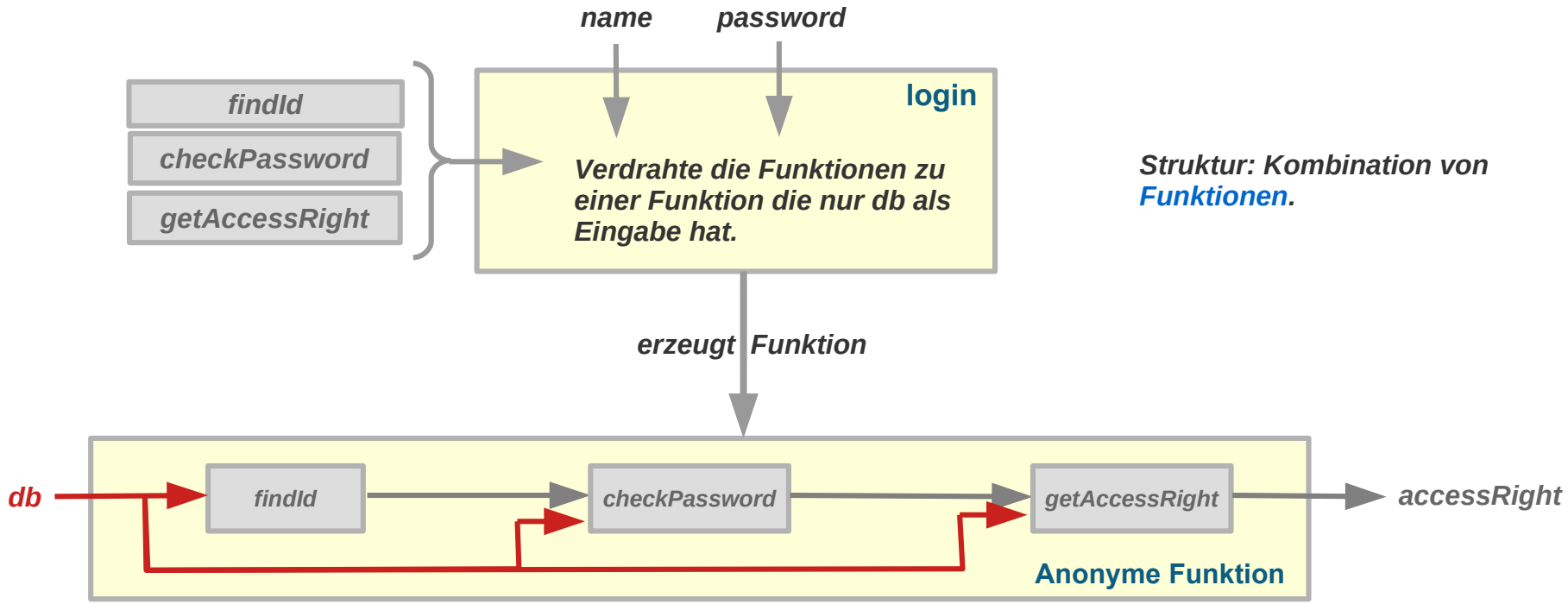
Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Ziel: *login* erzeugt aus *findId*, *checkPassword* und *getAccessRight,* sowie *Passwort* und *Name* eine Funktion vom Typ:

DB => AccessRight

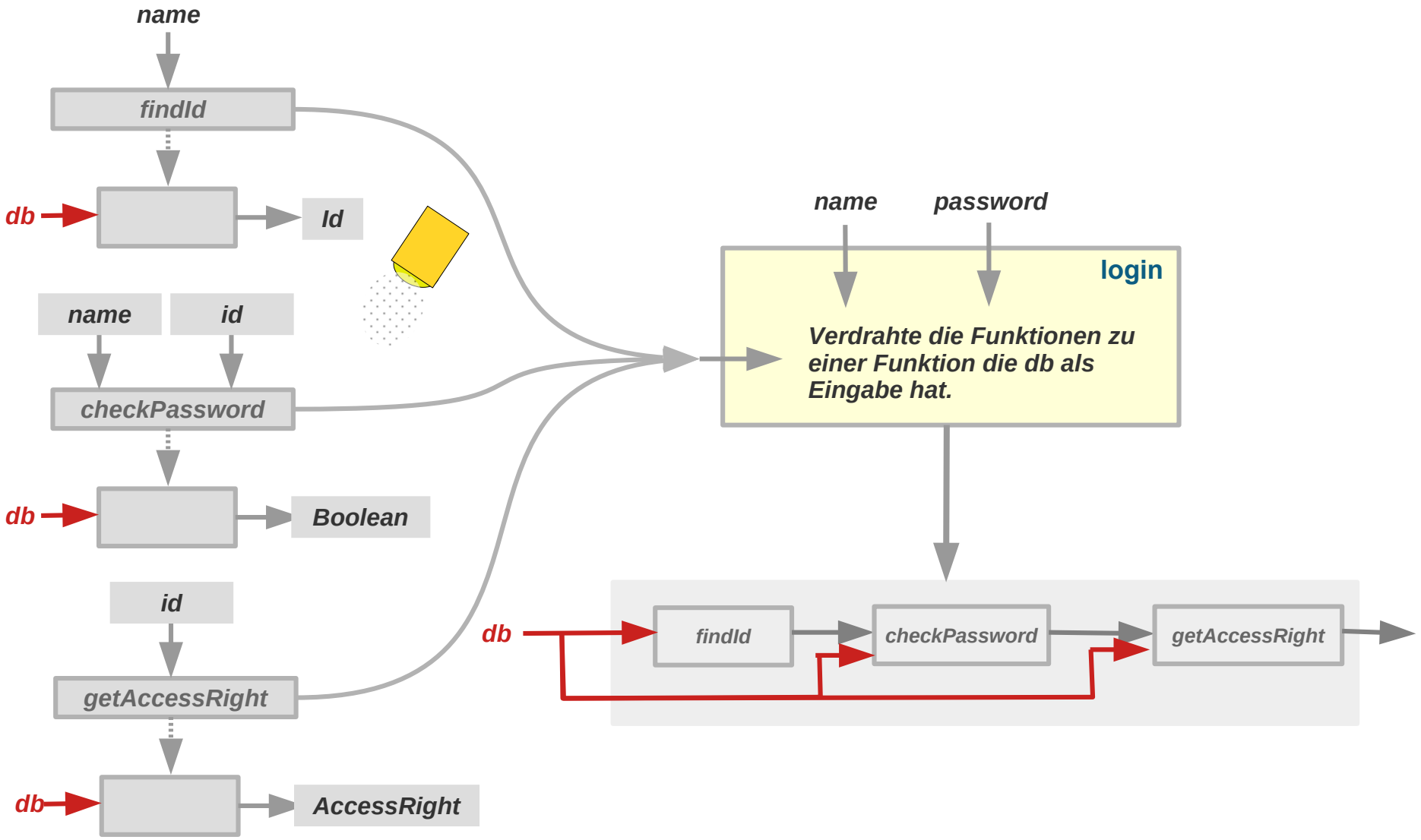
Vorteil / gewünschter Effekt: In *login* werden Funktionen miteinander kombiniert, das Argument dieser Funktionen db kann unerwähnt bleiben. Es muss nicht explizit herum geschleppt werden.



Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Curry: ***findId, checkPassword, getAccessRight werden „gecurryt“***



Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Curry: *findId, checkPassword, getAccessRight werden „gecurryt“*

```
def findId(name: String): DB => Int =
  db => db.getId(name).get

def checkPassword(id: Int, password: String): DB => Boolean =
  db => db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }

def getAccessRight(id: Int): DB => AccessRight =
  db => db.getAccessRight(id).get


def login(name: String, password: String): DB => AccessRight =
  db => {
    val id = findId(name)(db)
    val pwOk = checkPassword(id, password)(db)
    if (pwOk) getAccessRight(id)(db)
    else AccessRight.NoAccess
  }
```

Hmm, Currying bringt auch nicht viel. Hier wird immer noch mit db herumhantiert.

Von „Funktionsverdrahtung“ noch keine Spur.

Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Curry + explizite Verknüpfung von Funktion

Zwei äquivalente Formulierungen

```
def login(name: String, password: String): DB => AccessRight =
  db => {
    val id = findId(name)(db)
    val pwOk = checkPassword(id, password)(db)
    if (pwOk) getAccessRight(id)(db)
    else AccessRight.NoAccess
  }
```

mit val-Definitionen

≡

```
  def login(name: String, password: String): DB => AccessRight = db =>
    ((id: Int) =>
      ((pwOk: Boolean) =>
        if (pwOk) getAccessRight(id)(db) else AccessRight.NoAccess
        )(checkPassword(id, password)(db))
      )(findId(name)(db))
```

val-Definitionen zu Parameterübergaben aufgelöst

Verknüpfung von Funktionen

Funktionen sind Funktoren (siehe Foliensatz 4)

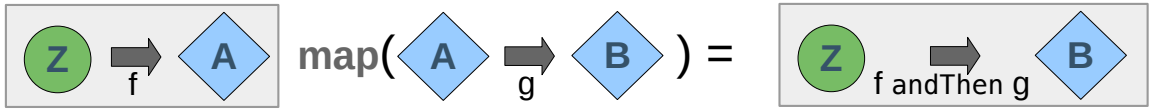
Kontextabhängige Werte sind vom Typ Kontext => Wert, also Funktionen

Funktionen mit fixiertem Definitionsbereich (hier Z genannt) sind Funktoren

map ~ Funktionsverknüpfung

```
type ZTo[T] = Z => T

given Functor[ZTo] with {
  extension[A, B] (f: ZTo[A]) def map(g: A => B): ZTo[B] =
    f andThen g
}
```

Z ist der Typ des Kontexts



Exkurs: Verknüpfung von Funktionen

Semimonaden sind Monaden ohne pure.

Funktionen sind Semimonaden

Funktionen mit fixiertem Definitionsbereich (hier Z genannt) sind Semimonaden mit folgendem flatMap: *Hintereinander-Ausführung mit weitergereichtem z*

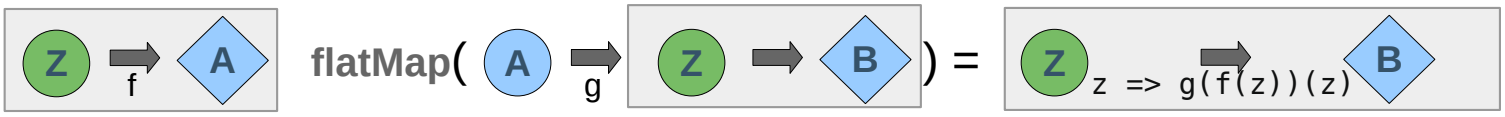
```
given SemiMonad[ZTo] with {
  extension[A, B] (f: ZTo[A]){
    def map(g: A => B): ZTo[B] =
      f andThen g
    def flatMap(g: A => ZTo[B]): ZTo[B] = z =>  g(f(z))(z)
  }
}
```

≡

```
given SemiMonad[ZTo] with {
  extension[A, B] (f: ZTo[A]){
    def map(g: A => B): ZTo[B] =
      f andThen g
    def flatMap(g: A => ZTo[B]): ZTo[B] = z => {
      val v1 = f(z)
      val v2 = g(v1)
      v2(z)
    }
  }
}
```

FlatMap: Erst f, dann g mit dessen Resultat und weiter mit dem gleichen z (Z ist irgendein fixierter Typ)

Das Gleiche, (eventuell) etwas übersichtlicher mit val-Definitionen



Verkett-/iterier-bare Kontext- (Z-) abhängige Aktionen!

Reader-Monade

Reader sind Funktionen, Funktionen sind Monaden

```
class Reader[Z, T](val f: Z => T) {
  def apply(z: Z) = f(z)
}


type DBReader[T] = Reader[DB, T]

given Monad[DBReader] with {
  def pure[A](a: A): DBReader[A] = Reader(db => a)
  extension[A, B] (dbReader: DBReader[A]) {
    def flatMap(g: A => DBReader[B]): DBReader[B] =
      Reader(z => {
        val v1 = dbReader.f(z)
        val v2 = g(v1)
        v2.f(z)
      } )
    override def map(g: A => B) = Reader( dbReader.f andThen g )
  }
}
```

Reader sind Funktionen.
Zum besseren Handling hier in einer Klasse gekapselt.

Funktionen mit festem Definitionsbereich sind Monaden.

Reader-Monaden

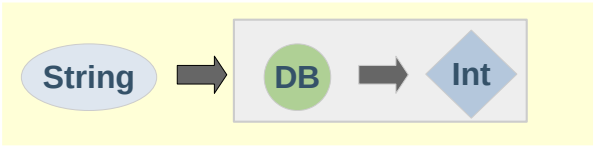
Beispiel Login-Daten prüfen

Die Hilfsfunktionen sind vom Typ

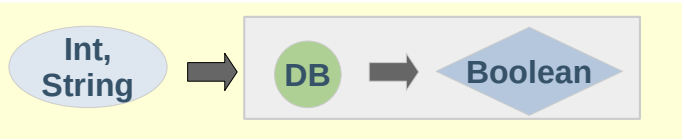
A => DBReader

und damit kompatibel mit flatMap

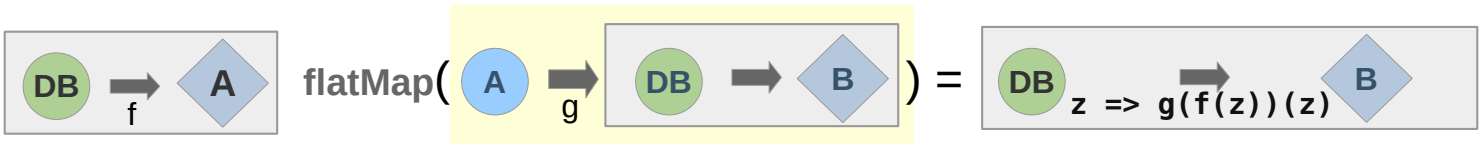
```
def findId(name: String): DBReader[Int] =
    Reader( (db: DB) => db.getId(name).get )
```




```
def checkPassword[R[_]: Monad](id: Int, password: String):
  DBReader[Boolean] =
  Reader( (db:DB) => db.getPassword(id) match {
      case Some(pw) if (pw == password) => true
      case _ => false
    }
  )
```




```
def getAccessRight[R[_]: Monad](id: Int): DBReader[AccessRight] =
  Reader( (db:DB) => db.getAccessRight(id).get)
```



Beispiel Login-Daten prüfen

Schritt 4) login mit DBReader als Exemplar der Typklasse Monade

```
def login(name: String, password: String): DBReader[AccessRight] =
  findId(name) flatMap(
    (id:Int) => checkPassword(id, password) flatMap(
      (pwOk: Boolean) =>
        if (pwOk) getAccessRight(id)
        else Monad[DBReader].pure(AccessRight.NoAccess)
    )
  )
```

login definiert mit flatMap

```
def login(name: String, password: String): DBReader[AccessRight] =
  for ( id <- findId(name);
        pwOk <- checkPassword(id, password);
        res <- (if (pwOk) getAccessRight(id)
                else Monad[DBReader].pure(AccessRight.NoAccess))
      ) yield res
```

… und mit for-Comprehension

Reader als monadische Klasse

DBReader lässt sich natürlich auch gleich als monadische Klasse definieren, die Definition einer Monaden-Instanz ist dann unnötig

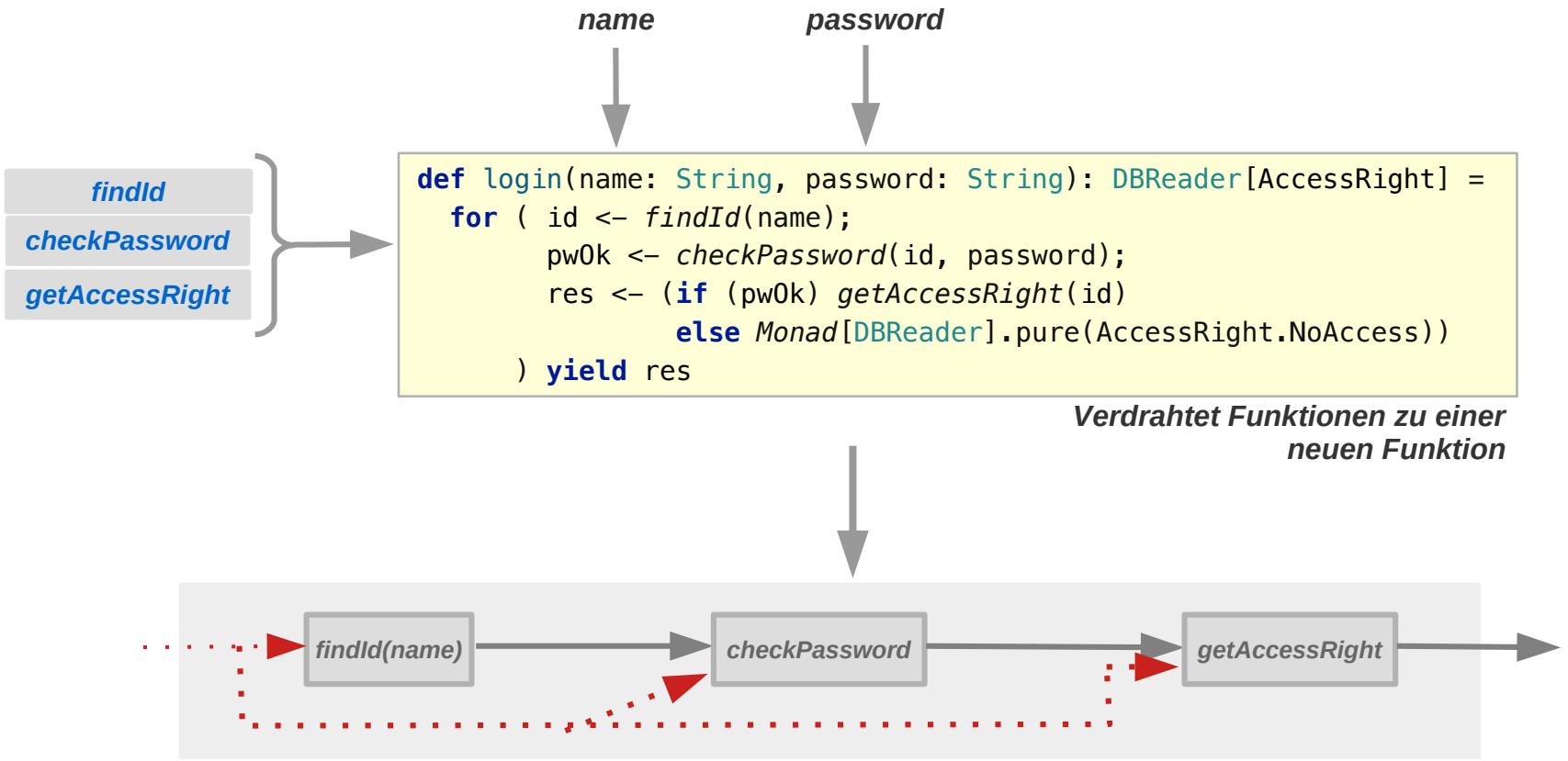
```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z => g(f(z)) (z) )
}

def findId(name: String): Reader[DB, Int] =
  Reader( (db: DB) => db.getId(name).get )

def checkPassword(id: Int, password: String): Reader[DB, Boolean] =
  Reader( (db:DB) => db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }
  )

def getAccessRight(id: Int): Reader[DB, AccessRight] =
  Reader( (db:DB) => db.getAccessRight(id).get)

def login(name: String, password: String): Reader[DB, AccessRight] =
  for ( id <- findId(name);
        pwOk <- checkPassword(id, password);
        res <- (if (pwOk) getAccessRight(id)
                else Reader(AccessRight.NoAccess)))
    yield res
```

Reader-Monaden: Beispiel Login

Beispiel Login-Daten prüfen

Ziel erreicht: login ist eine Komposition von Funktionen

Das Funktionsargument db kommt in login nicht mehr vor: Es wird „völlig im Hintergrund“ weitergereicht. Funktionskombination statt Werteverarbeitung!



Reader-Monaden: Beispiel Terme auswerten

Beispiel Terme mit definierten Konstanten auswerten

Terme mit Konstanten auswerten

***Dependency* / Kontext: *Environment* mit Wert der Konstanten**

Naive Version aber schon ml gleich „ge-curry-t“:

```
enum Term {
  case Literal(v: Int)
  case Const(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]

def eval(term: Term): Env => Int =
  env => term match {
    case Literal(v) => v
    case Const(n) => env(n)
    case Add(t1, t2)  => eval(t1)(env) + eval(t2)(env)
    case Sub(t1, t2)  => eval(t1)(env) - eval(t2)(env)
    case Mult(t1, t2) => eval(t1)(env) * eval(t2)(env)
    case Div(t1, t2)  => eval(t1)(env) / eval(t2)(env)
  }

val term: Term = Add(Literal(18), Div(Mult(Literal(12), Literal(4)), Const("two")))
val termValue = eval(term)(Map("two" -> 2)) // 42
```

Env wird in jedem Aufruf weitergereicht.

Reader-Monaden: Beispiel Terme auswerten

Beispiel Terme mit definierten Konstanten auswerten

Terme mit Konstanten auswerten

mit Reader-Klasse:

```
type EnvReader[T] = Reader[Env, T]

def eval(term: Term): EnvReader[Int] = term match {
  case Literal(v) => Reader(v)

  case Const(n) => Reader(env => env(n))

  case Add(t1, t2)  =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l+r

  case Sub(t1, t2)  =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l-r

  case Mult(t1, t2)  =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l*r

  case Div(t1, t2)  =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l/r
}
```

```
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) )
}
```

Reader-Monaden: Beispiel Terme auswerten

Beispiel Terme mit definierten Konstanten auswerten

Terme mit ~~Konstanten~~ Variablen auswerten

Erweiterung auf „Programme“:

```
case class Assign(varName: String, value: Term)
type Prog = List[Assign]

 def eval(exp: Term): Reader[Env, Int] =
   exp match {
     case Literal(v) => Reader(v)
     case Variable(n) => Reader(env => env(n))
     case Add(t1, t2) =>
       for (v1 <- eval(t1);
            v2 <- eval(t2))
         yield v1 + v2

   ...
 }
```

```
def executeAssgin(assign: Assign): Env => Env =
  assign match {
    case Assign(varName, exp) =>
      env => env + (varName -> eval(exp)(env))
  }


def execute(prog: Prog): Env => Env =
  prog.foldLeft((env: Env) => env)( (acc, assign) =>
    acc andThen executeAssgin(assign)
  )
```

```
val program = List(
  Assign("x", Literal(2)),
  Assign("y", Mult(Variable("x"), Variable("x"))),
  Assign("y", Add(Variable("y"), Variable("y"))),
  Assign("r", Add(Variable("x"), Div(Mult(Variable("y"), Literal(10)), Literal(2))))
)

val result = execute(program)(Map())("r") // 42
```

Ein Beispiel-Programm:

```
x = 2
y = x*x
y = y + y
r = x+y*10/2
```

Writer-Monade

Berechnung in einem Kontext – der als Datensenke agiert

Bei einer Writer-Monade stellt der Kontext eine Datensenke zur Verfügung

Mit einer Writer-Monade

- **strukturiert man darum Berechnungen, die von einem Daten-konsumierenden Kontext abhängen.**
- **Sie sind damit ein (weiteres) Idiom / Muster für (funktionale) *Dependency Injection***

Writer-Monade

Beispiel Logging

Kontext / Dependency : Logging Mechnanismus / Konsument von Log-Informationen

Nicht-funktionale / imperative Version
Logger verändert sich bei Benutzung

```
trait Logger {
  def log(msg: String): Unit // Seiteneffekt
  def getLogs(): String
}
```

In der imperativen Version werden Logs in einem veränderlichen Objekt vom Typ Logger aufbewahrt.

```
object logger extends Logger {
  private var logBuffer: StringBuffer = new StringBuffer
  def log(msg: String): Unit = logBuffer.append(msg+"\n")
  def getLogs(): String = logBuffer.toString
}
```

Ein sehr einfacher Logger.

```
def f(x:Int): String = {
  logger.log(s"calling f($x)")
  (x+2).toString
}

def g(s:String): Int = {
  logger.log(s"calling g($s)")
  2 * s.length
}

def h(i: Int): Boolean = {
  logger.log(s"calling h($i)")
  i % 2 == 0
}
```

```
def loggedComputation(x: Int): Boolean = {
  val y = f(x)
  val z = g(y)
  h(z)
}

val result = loggedComputation(2)
val logs = logger.getLogs()
```

Writer-Monade

Beispiel Logging

Kontext / Dependency : Logging Mechnanismus / Konsument von Log-Informationen

Funktionale Version: Veränderungen gibt es nicht: Veränderungen werden als Werte modelliert Die in neuer Version expliziert herum gereicht werden.

```
case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}
```

Konkrete Wert-Klasse. Auf eine funktionale Datenabstraktion (siehe Foliensatz 1) wird der Einfachheit halber verzchtet.

```
def f(x:Int): (String, Log) =
  ((x+2).toString, Log(s"calling f($x)"))
```

```
def g(s:String): (Int, Log) =
  (2 * s.length, Log(s"calling g($s)"))
```

Die Funktionen haben ein zusätzliches Ergebnis: die von ihnen produzierten Logs.

Hinweis: Reader-Monaden behandeln ein zusätzliches Argument.

```
def h(i: Int): (Boolean, Log) =
  (i % 2 == 0, Log(s"calling h($i)"))
```

```
def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y)
  val (u, log3) = h(z)
  (u, log1.conc(log2).conc(log3))
}
```

Lästiges Herum-Hantieren mit den zusätzlichen (Log-) Ergebnissen.

Writer-Monade

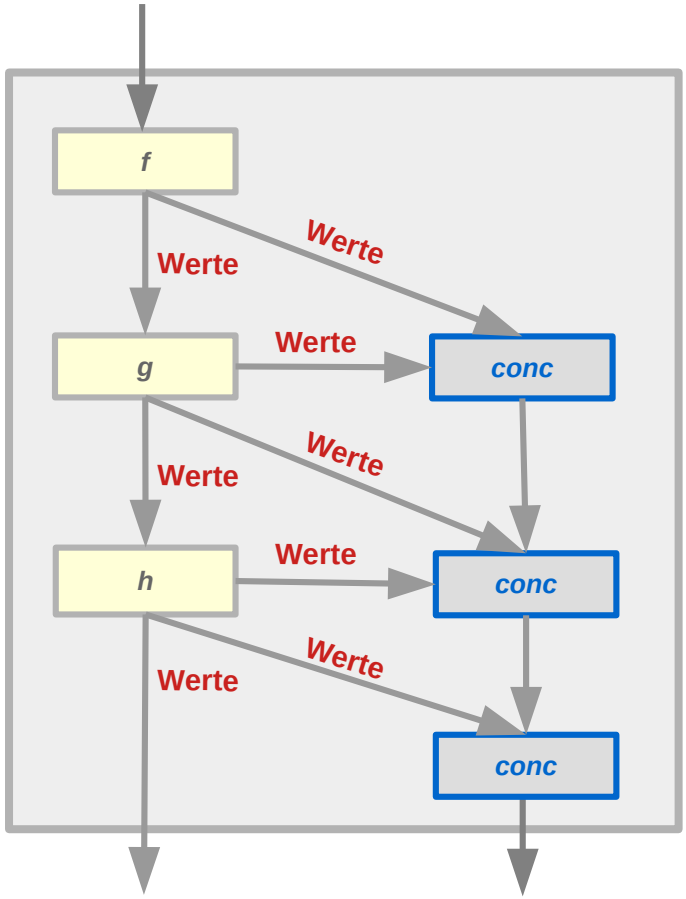
Beispiel Logging

Ziel: *loggedComputation* erzeugt aus *f*, *g* und *h* eine Funktion vom Typ:

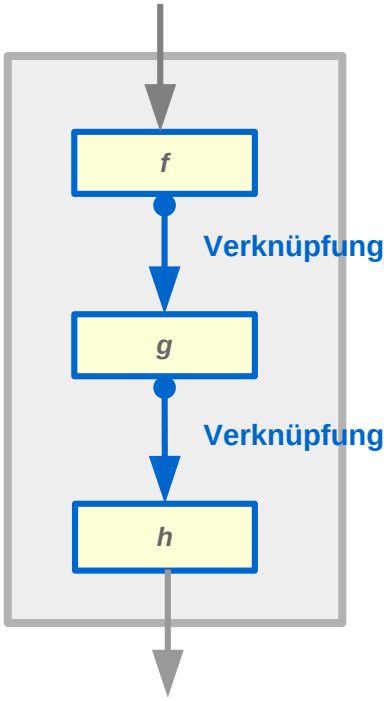
Int => (Boolean, Log)

Vorteil / gewünschter Effekt: In *loggedComputation* werden Funktionen geeignet (wie?) kombiniert. Das Log-Ergebnis dieser Funktionen kann dann unerwähnt bleiben. (Muss nicht explizit herum geschleppt werden.)

Explizite Verarbeitung der Ergebnisse der Funktionen f, g, h



?



Implizite Verarbeitung der beiden Ergebnisse von f, g, h durch die Kombination der Funktionen (nicht der Funktionsergebnisse!) f, g und h.

Writer-Monade

Beispiel Logging

Das Herum-Hantieren mit Logs wegfallen soll. Logged-Computation soll „kombinatorisch“ werden, d.h. statt

- **auf der Ebene der Werte, soll derAlgorithmus**
- **auf der Ebene von Funktionen / als Kombination von Funktionen**

formuliert werden.

Schritt 1: Umgang mit Logs vereinheitlichen

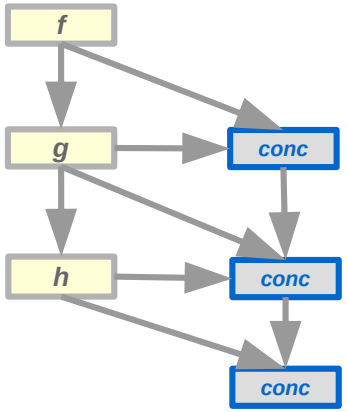
```
def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y)
  val (u, log3) = h(z)
  (u, log1.conc(log2).conc(log3))
}
```

Logs der Teilergebnisse werden am Schluss verarbeitet.

~

```
def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y).match {case (v, l) => (v, log1.conc(l))}
  h(z).match {case (v, l) => (v, log2.conc(l))}
}
```

Logs der Teilergebnisse in jedem Schritt verarbeitet.



Writer-Monade

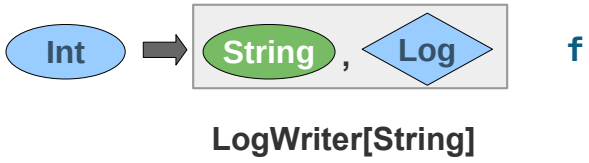
Beispiel Logging

Schritt 2 Beobachtung:

- Die Hilfsfunktionen sind vom Typ A => (B, Log)
- Mit einem Funktor F[•] = (•, Log)
  haben sie ein Typ der Form A => F[B]
  und sind damit „flatMap-kompatibel“
- Nennen wir den Typ F[•] = LogWriter[•] dann erhalten wir:

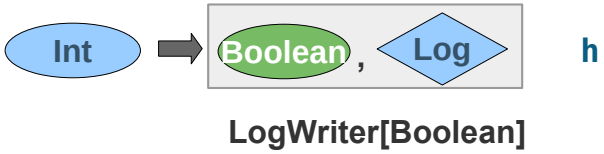
```
type LogWriter[A] = (A, Log)
```

```
def f: Int => LogWriter[String] = x =>
  ((x+2).toString, Log(s"calling f($x)"))
```




```
def g: String => LogWriter[Int] = s =>
  (2 * s.length, Log(s"calling g($s)"))
```

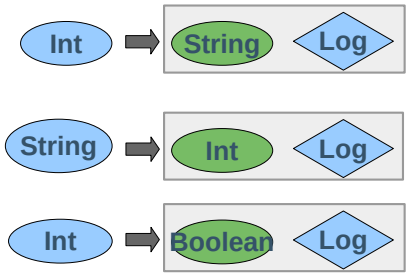

```
def h: Int => LogWriter[Boolean] = i =>
  (i % 2 == 0, Log(s"calling h($i)"))
```



Writer-Monade

Beispiel Logging

Schritt 2: Das passende flatMap ist offensichtlich:




```
case class LogWriter[A](a: A, log: Log) {
  def flatMap[B](f: A => LogWriter[B]): LogWriter[B] = {
    f(a) match {
      case LogWriter(v, l) =>
        LogWriter[B](v, log.conc(l))
    }
  }
}
```

```
def f: Int => LogWriter[String] = x =>
  ((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  (2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  (i % 2 == 0, Log(s"calling h($i)"))
```

```
def loggedComputation: Int => LogWriter[Boolean] = x =>
  f(x).flatMap(g).flatMap(h)
```

Writer-Monade

Beispiel Logging

Verallgemeinerung: Der Typ der Logs kann ein beliebiges Monoid sein

```
trait Monoid[T] {
  def unit: T
  def combine(x: T, y: T): T
}
```

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

case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}

given Monoid[Log] {
  def unit = Log("")
  def combine(l1:Log, l2:Log): Log = l1.conc(l2)
}
```

```
type LogWriter[A]= Writer[A, Log]

def f: Int => LogWriter[String] = x =>
  Writer((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  Writer(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  Writer(i % 2 == 0, Log(s"calling h($i)"))

def loggedComputation: Int => Writer[Boolean, Log] = x =>
  f(x).flatMap(g).flatMap(h)
```

Writer-Monade

Beispiel Logging

Mit For-Comprehension

```
case class Writer[A, W:Monoid](a: A, w: W) {
  def this(a: A) = this(a, summon[Monoid[W]].unit) // pure als Konstruktor
  def map[B](f: A => B): Writer[B, W] = Writer(f(a), w)
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
    val Writer(v, w1) = f(a)
    Writer[B, W](v, summon[Monoid[W]].combine(w, w1))
  }
}

type LogWriter[A]= Writer[A, Log]

def f: Int => LogWriter[String] = x =>
  Writer((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  Writer(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  Writer(i % 2 == 0, Log(s"calling h($i)"))

def loggedComputation: Int => Writer[Boolean, Log] = x =>
  for (a <- f(x);
       b <- g(a);
       c <- h(b))
    yield c
```

Writer komplett als Monade (monadische Klasse)

Beispiel Logging

In Typklassen-Notation (1)

```
case 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))
  }
}

case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}

type LogWriter[A] = Writer[A, Log]

given Monoid[Log] with {
  def unit = Log("")
  def combine(l1:Log, l2:Log): Log = l1.conc(l2)
}


given Monad[LogWriter] with {

  def pure[A](a: A): LogWriter[A] = Writer(a, Monoid[Log].unit)

  extension[A, B] (logWriter: LogWriter[A]) {
    def flatMap(f: A => LogWriter[B]): LogWriter[B] = {
      val lw = f(logWriter.value)

      Writer(lw.value, Monoid[Log].combine(logWriter.w, lw.w))
    }
  }
}
```

Beispiel Logging

In Typklassen-Notation (2)

```
def f: Int => LogWriter[String] = x =>
  Writer((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  Writer(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  Writer(i % 2 == 0, Log(s"calling h($i)"))

def loggedComputation(x: Int): LogWriter[Boolean] = {
  for (y <- f(x);
       z <- g(y);
       u <- h(z))
  yield u
}

val resultAndlog = loggedComputation(2)
val log = resultAndlog.w
```