Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Datentypen und Datenabstraktionen

- Datenabstraktion: Struktur oder Schnittstelle
- Funktionale Datenabstraktion mit existenziellen / abstrakten Typen
- ADTs und GADTs
- Typabstraktionen mit Typklassen
- Typrelationen und Phantomtypen

Datenabstraktion:
Typen als Verhaltens- oder Strukturmuster

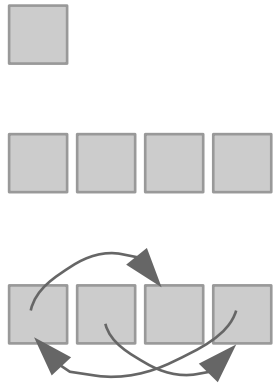
Typen: Verhalten oder Struktur von Werten

Datenstrukturen

Basis

Jede Datenstruktur in jeder Programmiersprache ist entweder

- **elementar / atomar, oder**
- **eine Sequenz von Daten (Array, Record, Objekt, ...), oder**
- **eine Verkettung (via Pointer) von Daten, oder**
- **Ein Kombination dieser Möglichkeiten**



Auf diesen drei schmalen Füßchen werden himmelhohe Abstraktionsgebäude errichtet

Typen: Verhalten oder Struktur von Werten

Datenstrukturen

Funktionale Datenstrukturen

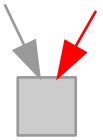
Datenstrukturen in der funktionalen Programmierung sind beschränkt

- **Persistenz: Daten werden nie geändert**

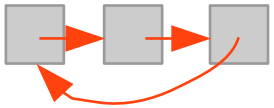
  Arbeite mit (partiellen) Kopien statt mit Modifikationen

- **Beschränkte Verkettung**
  - **Sharing**

    nur intern: Auf ein Datum zeigt (konzeptionell) immer nur maximal ein Zeiger
    Konsequenz: Keine Unterscheidung zwischen „das Selbe“ und „das Gleiche“
  - **Verkettete Strukturen sind strikt hierarchisch (keine Zyklen)**



Sharing: Nur als Optimierung erlaubt für die Anwendung immer unsichtbar



Zyklen: Verboten

Typen: Verhalten oder Struktur von Werten

Typen: Zwei Sichtweisen

Interface-Sicht: Datentyp ~ Verhalten

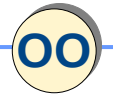
Katalogisieren von Daten auf Basis ihres (äußeren) Verhaltens / ihrer Schnittstelle

Populär im imperativen / OO – Umfeld

Struktur-Sicht: Datentyp ~ Struktur von Daten

Katalogisieren von Daten auf Basis ihres (inneren) Aufbaus

populär im funktionalen Umfeld



Typen – Interface Sicht (aka *Information Hiding*)

Datentyp ~ Verhalten

Typ: Katalogisieren von Daten auf Basis ihres (äußeren) Verhaltens / ihrer Schnittstelle

- **Abstrakter Datentyp (ADT): Interface mit unterschiedlichen Implementierungen**
- **Populär / sinnvoll für prozedurale- /objektorientierte Programmierung**
- **Beispiel Stack:**

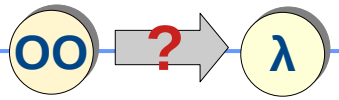
```scala
trait Stack[A] {
  def push(a: A): Unit
  def pop(): A
}
```

Interface definiert Verwendungsmöglichkeiten.

```scala
class ListStack[A] extends Stack[A] {
  import scala.collection.mutable.ListBuffer

  private val content = new ListBuffer[A]()

  override def push(a: A): Unit = content.prepend(a)
  override def pop(): A = content.remove(0)
}
```

Implementierung: mit geheimen Interna, und (darum) jederzeit austauschbar gegen eine mit anderen Interna.

Typen: Verhalten oder Struktur von Werten



Typen – Interface Sicht (aka *Information Hiding*)

Datentyp ~ Verhalten

Typ: Katalogisieren von Daten auf Basis ihres (äußeren) Verhaltens / ihrer Schnittstelle

- **geht auch funktional**
- **Beispiel Stack:**

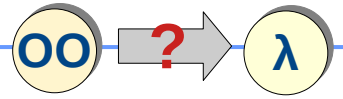
```
trait Stack[A] {
  def push(a: A): Stack[A]
  def pop(): (A, Stack[A])
}
```

Interface definiert Verwendungsmöglichkeiten. Hier: funktionale Verwendung.

```
class ListStack[A](
    private val content: List[A] = List()
  ) extends Stack[A] {

  override def push(a: A): Stack[A] =
    ListStack(a :: content)

  override def pop(): (A, Stack[A]) =
    (content.head, ListStack(content.tail))
}
```

Die funktionale Verwendung wird schnell unhandlich: Operationen liefern Ergebnispaar.

Typen: Verhalten oder Struktur von Werten

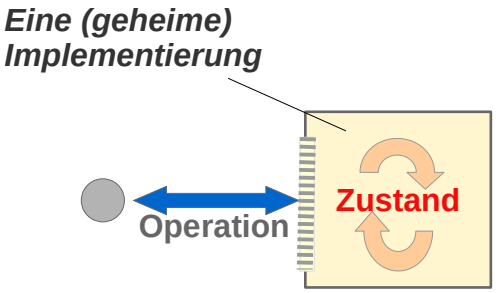


Typen – Interface Sicht und das funktionale Paradigma

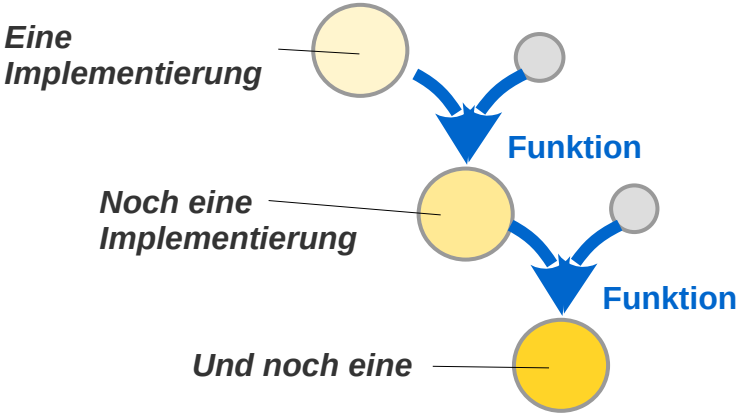
Datentyp ~ Verhalten

Typ: Katalogisieren von Daten auf Basis ihres (äußeren) Verhaltens / ihrer Schnittstelle

- **geht auch funktional**
- **ist aber i.A. (in dieser einfachen Art) nicht praktisch / sinnvoll**
  Werte erzeugende Funktionen sind üblich.
  Operationen, die an der Schnittstelle veränderlicher Objekte arbeiten, gibt es nicht.



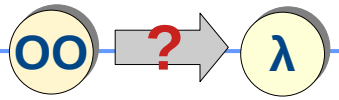
OO / Imperativ: Interface beschreibt Schnittstelle veränderlicher Objekte



Können die Implementierungen geheim bleiben?

Die Interfacesicht ist nicht (einfach) in den Kontext der funktionalen Pro-grammierung übertragbar!

Typen: Verhalten oder Struktur von Werten



Typen – Interface Sicht (aka *Information Hiding*)

Datentyp ~ Verhalten

Typ: Katalogisieren von Daten auf Basis ihres (äußeren) Verhaltens / ihrer Schnittstelle

- **geht auch funktional – aber nicht so einfach / und nicht sinnvoll**
- **Beispiel Komplexe Zahlen**

```
trait Complex {
  def add (other: Complex): Complex
}

case class CartComplex(val x: Double, y: Double) extends Complex {
  def add (other: Complex): Complex =
     if (other.isInstanceOf[CartComplex])
          new CartComplex(
               x +
               other.asInstanceOf[CartComplex].x,
               y +
               other.asInstanceOf[CartComplex].y)
     else ??? hmm, was jetzt ???
}

case class PolarComplex(val r: Double, phi: Double) extends Complex {
  def add (other: Complex): Complex = ???
}
```

Information Hiding kann nicht so einfach von der OO- in die funktionale Welt übertragen werden. Aber es geht.

Existenzielle Typen

All-Quantor auf Typen

Eine generische Funktion wie etwa

```
def length[A](lst: List[A]): Int = lst.length
```

hat den Typ

```
∀A.List[A] => Int
```

Für alle Typen `A` hat die Funktion den Typ `List[A] => Int`

Existenz-Quantor auf Typen

Kann es etwas geben mit dem Typ

```
∃A. ...
```

Es gibt einen Typ `A` für den ….

Existenzielle Typen

Existenzielle-Typen

Typen die mit dem Existenz-Quantor gebildet werden.

Einsatz-Beispiel: Abstrakter Datentyp funktional / funktionale Datenabstraktion

Gesucht Funktionale Version der Datenabstraktion Stack:

```
trait StackI {
  def push(x: Int): Unit
  def pop(): Int
}

class ListStackI extends StackI {
  private val stackImpl: ListBuffer[Int] = new ListBuffer[Int]()
  override def push(x: Int): Unit = stackImpl.prepend(x)
  override def pop(): Int = stackImpl.remove(0)
}
```

Schnittstelle

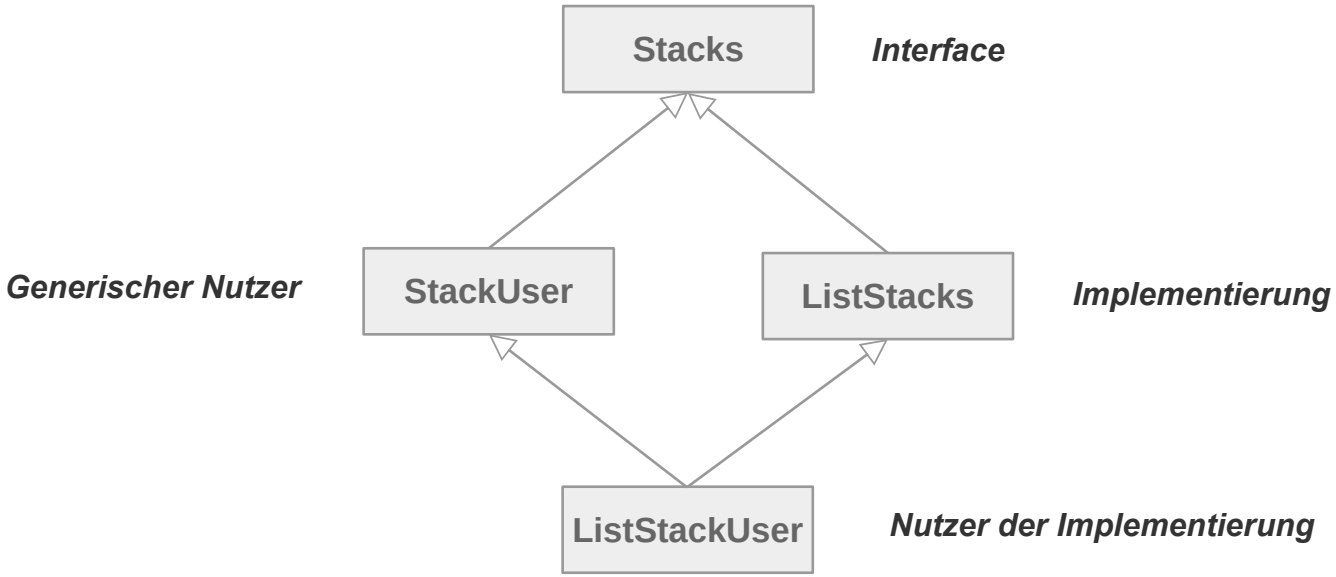
Implementierung mit geheimer Datenrepräsentation

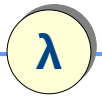
Datenabstraktion im imperativen / OO Kontext: alles passt

Datenabstraktion – funktional

Einsatz existenzielle Typen: Funktionaler Abstrakter Datentyp

Übersicht





Datenabstraktion – funktional

Einsatz-Beispiel Existenzielle Typen: Funktionaler Abstrakter Datentyp Stack (1)

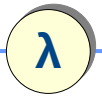
```
trait Stacks {

  trait StackF[StackRep] {
    def push(x: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  type Stack = StackF[StackRep] forSome { type StackRep }

  def emptyStack: Stack

}
```

Schnittstelle

***Existenzieller Typ** ab Scala 3 nicht mehr unterstützt.*

```
trait StackUser extends Stacks {

  def useStack(): Unit = {
    val s0: Stack = emptyStack
    val s1 = s0.push(1)
    val s2 = s1.push(2)
    val (v1, s3) = s2.pop
    val (v2, s4) = s3.pop

    // keine Chance zum Zugriff
    //auf die Implementierung des Stacks

    println(v1)
    println(v2)
  }

}
```

Ein Nutzer der Schnittstelle



Datenabstraktion – funktional

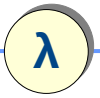
Einsatz existenzielle Typen: Funktionaler Abstrakter Datentyp Stack (2)

```
class ListStacks extends Stacks {

  private class ListStackF(private val rep: List[Int]) extends StackF[List[Int]] {
    def emptyStack: StackF[List[Int]] =
      new ListStackF(List())

    def push(x: Int): StackF[List[Int]] =
      new ListStackF(x :: this.rep)

    def pop: (Int, StackF[List[Int]]) =
      (rep.head, new ListStackF(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStackF(List())

}
```

Implementierung der Stacks mit Listen

```
object Main {

  object ListStackUser extends ListStacks with StackUser

  def main(args: Array[String]): Unit = {
    ListStackUser.useStack()
  }

}
```

Ein `spezifischer Nutzer' der Implementierung



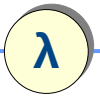
Datenabstraktion – funktional

Einsatz existenzielle Typen

Problem: existenzielle Typen werden in Scala 3 nicht mehr unterstützt

Ersatz 1: abstrakter Typ

```
trait Stacks {

  trait StackF[StackRep] {
    def push(x: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  //type Stack = StackF[StackRep] forSome { type StackRep }
  protected type SomeStackRep
  type Stack = StackF[SomeStackRep]

  def emptyStack: Stack
}
```



Datenabstraktion – funktional

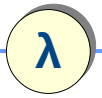
Einsatz existenzielle Typen

Problem: existenzielle Typen werden in Scala 3 nicht mehr unterstützt

Ersatz 1: abstrakter Typ

```
class ListStacks extends Stacks {

  override type SomeStackRep = List[Int]

  private class ListStack(private val rep: List[Int]) extends StackF[SomeStackRep] {
    def emptyStack: StackF[SomeStackRep] =
      new ListStack(List())

    def push(x: Int): StackF[SomeStackRep] =
      new ListStack(x :: this.rep)

    def pop: (Int, StackF[SomeStackRep]) =
      (rep.head, new ListStack(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStack(List())

}
```

Typen: Verhalten oder Struktur von Werten



Datenabstraktion – funktional

Einsatz existenzielle Typen

Problem: existenzielle Typen werden in Scala 3 nicht mehr unterstützt

Ersatz 2: *Wildcard*-Typ

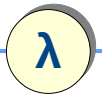
```
type Stack = StackF[StackRep] forSome { type StackRep }
```

```
Existential types are no longer supported
- use a wildcard or dependent type instead
```

```scala
trait Stacks {

  trait StackF[StackRep] {
    def push(a: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  type Stack = StackF[?]

  def emptyStack: Stack

}
```

```scala
class ListStacks extends Stacks {

  private class ListStack(private val rep: List[Int]) extends StackF[List[Int]]
{
    def emptyStack: StackF[List[Int]] =
      new ListStack(List())

    def push(x: Int): StackF[List[Int]] =
      new ListStack(x :: this.rep)

    def pop: (Int, StackF[List[Int]]) =
      (rep.head, new ListStack(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStack(List())
}
```

Typen: Verhalten oder Struktur von Werten



Datenabstraktion – funktional

Einsatz existenzielle Typen

Problem: existenzielle Typen werden in Scala 3 nicht mehr unterstützt

Ersatz 3: *Dependent Type*

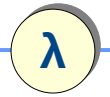
***Dependent Type* : Typ der von einem Wert abhängig ist.**

```
trait Stacks {

  trait StackF { // Schnittstelle
    type StackRep
    val stackRep: StackRep
    def push(a: Int): StackF
    def pop: (Int, StackF)
  }

  // dependent type: der Wert rep bestimmt den Typ Rep
  abstract class StackBase[Rep](rep: Rep) extends StackF {
    type StackRep = Rep
    val stackRep = rep
  }

  def emptyStack: StackF
}
```

```
type Stack = StackF[StackRep] forSome { type StackRep }

Existential types are no longer supported
- use a wildcard or dependent type instead
```

```
class ListStacks extends Stacks {

  case class Stack(rep: List[Int]) extends StackBase(rep) {

    def push(x: Int): StackF =
      new Stack(x :: rep)

    def pop: (Int, Stack) =
      (rep.head, new Stack(rep.tail))
  }

  override def emptyStack = Stack(List())
}
```

ADTs und GADTs:
Typen als Strukturmuster



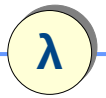
Typen – Struktur-Sicht

Datentyp ~ Struktur der Daten

Typ: Katalogisieren von Daten auf Basis ihrer (inneren) Struktur

- **Beobachtung:**
  Alle funktionalen (!) Datenstrukturen (keine Zyklen, keine zwei Zeiger auf ein Datum) können in ihrer Essenz als
  - **Zusammensetzen aus Einfacherem (Typ-Produkt) und**
  - **Auswahl aus Alternativen (Typ-Summe)**

  gedeutet interpretiert werden.
- **Beispiel:**
  - **Liste: leer oder (erstes Element und Rest)**
  - **Baum: Blatt oder (Wert und linker und rechter Unterbaum)**
  - **…**



ADT – algebraischer Datentyp: ein Struktursicht auf Typen

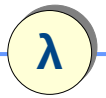
Algebraischer Datentyp

Klasse rein funktionaler Datenstrukturen einer bestimmten Art / Struktur
Datentyp ~ Eine Form von (Daten-) Strukturen

In der funktionalen Programmierung eher angemessen als die Interface-Sicht

ADT: Definition von Strukturen mit

- **Summe / Alternative,**
- **Produkt / Kombination**
- **Rekursion**



ADT – algebraischer Datentyp: ein Struktursicht auf Typen

Algebraischer Datentyp

Strukturen fester endlicher oder unbestimmter, potentiell unendlicher Größe

- **Endlich: (Typ-) Polynome**
  - **Typ-Summen und**
  - **Typ-Produkte**
- **(Potentiell) Unendlich: (Typ-) Gleichungen**
  Rekursiv definierte Typen

Typ-Produkte und Typ-Summen

Produkt-Typen

Jede (vernünftige) Programmiersprache erlaubt die Definition von Produkttypen und Werten

Beispiel C

Paar als struct

```c
typedef struct Pair {
  int left;
  int right;
} Pair;
```

Paar als Array

```c
typedef int Pair[2];
```

```c
Pair pair = {1, 2};
```

Beispiel Scala

```scala
case class Pair(left: Int, right: Int)
```

```scala
val pair = Pair(1, 2)
```

Beispiel Swift

```swift
class Pair {
    let left: Int, right: Int
    init (left: Int, right: Int) {
        self.left = left; self.right = right
    }
}
```

```swift
let pair: Pair = Pair(left: 1, right: 2)
```

Beispiel Haskell

```haskell
data Pair = Pair Int Int
```

```haskell
pair :: Pair
pair = Pair 1 2
```

Typdefinition | **Wertdefinition**

Produkt-Typen

Produkt-Typen sind „Und-Typen“:

ein Wert von T_1 und ein Wert von T_2 … ergibt einen Wert vom Produkt-Typ P

Produkttypen gibt es in vielfältiger Form in den Programmiersprachen.
Manche Sprachen unterstützen mehrere Formen

- **Tupel**
- **Objekt**
- **Struct**
- **…**

Die einfachste Variante sind die Tupel

$P = (T_1 \times \cdots \times T_n)$

Summen-Typen

Summen-Typen sind „Oder-Typen“:

ein Wert von T_1 oder ein Wert von T_2 … ergibt einen Wert vom Summen-Typ S

Summentypen sind verbreitet in den Programmiersprachen zu finden.

Manche Sprachen unterstützen mehrere Formen

- **Union-Typen in C / C++**
- **Aufzählungstypen**
- **Implizit: Alle Klassen mit der gleichen Basis-Klasse**
- **…**

Die einfachste (kanonische) Variante sind Aufzählungen

$S = (T_1 + \bullet\bullet\bullet + T_n)$ üblich ist auch | statt + : $S = (T_1 | \bullet\bullet\bullet | T_n)$

***Tagged Union*: Die Komponenten einer Summe sind immer identifizierbar: Es es ist immer klar zu welcher Alternative ein Wert gehört.**

Man kann sich vorstellen, dass die Werte mit einer Typ-Markierung versehen sind,

Summen-Typen

Beispiel OO-Sprache (Java / Scala): Vererbungsrelation als Typsumme

```
abstract class Animal
case object Cow extends Animal
case object Tiger extends Animal
```

*Ein Tier ist eine Kuh **oder** ein Tiger*

Summen-Typen

Beispiel C: Union

```c
union MyUnion { // Typdefinition. Hier
     int i;    // Entweder ein int
     char c;   // Oder ein char
};

union MyUnion unionVar = { 42 }; // Variablendefinition

// Zugriffe auf die selben Bits
printf("%i, %c\n", unionVar.i, unionVar.c);
unionVar.c++;
printf("%i, %c\n", unionVar.i, unionVar.c);
```

```
42, *
43, +
```

40	0x28	050	(
41	0x29	051	)
42	0x2A	052	*
43	0x2B	053	+

Ausschnitt ASCII-Tabelle

Das ist kein Bug sondern ein Feature: Die Sprache C wurde für die Systemprogrammierung konzipiert, nicht für die Anwendungsentwicklung.

Summen-Typen

Beispiel Scala: Typ-Summe als Eum

```
enum Animal {
  case Cow
  case Tiger
}
```

```
enum MyUnion {
  case IntU(i: Int)
  case CharV(c: Char)
}
```

Summen-Typen

Beispiel OO-Sprache (Java / Scala): Vererbungsrelation als Typsumme

```java
interface MyUnion {};

static class Variante1 implements MyUnion {
    Variante1(int i) { this.i = i; }
    int i;
}

static class Variante2 implements MyUnion {
    Variante2(char c) { this.c = c; }
    char c;
}

public static void main(String[] args) {
    MyUnion unionVar = new Variante1(42);
    System.out.println(((Variante1)unionVar).i);
    System.out.println(((Variante2)unionVar).c);
    ((Variante1)unionVar).i++;
    System.out.println(((Variante1)unionVar).i);
    System.out.println(((Variante2)unionVar).c);
}
```

Java

```
42
Exception in thread "main"
java.lang.ClassCastException
```

```scala
trait MyUnion {}
case class Variante1(var i: Int) extends MyUnion
case class Variante2(var c: Char) extends MyUnion

val unionVar = Variante1(42)
println(unionVar.asInstanceOf[Variante1].i)
println(unionVar.asInstanceOf[Variante2].c)
unionVar.asInstanceOf[Variante1].i += 1
println(unionVar.asInstanceOf[Variante1].i)
println(unionVar.asInstanceOf[Variante2].c)
```

Scala

Das ist ein Bug: Die Sprache Java wurde für die Anwendungsentwicklung konzipiert, so etwas sollte zu einem Übersetzungsfehler nicht erst zu einem Laufzeitfehler führen.

Summen- und Produkt-Typen

Nothing = leerer Summen-Typ

Die leere Summe, der Typ Nothing hat keinen „Bewohner“ (Exemplar)

Aus 0 Alternativen kann man nichts auswählen

Unit = leerer Produkt-Typ

Die leere Produkt, der Typ Unit hat genau einen „Bewohner“

Mit nichts kann man eine Schachtel auf genau eine Art füllen: es kommt nichts hinein

Option = Unit + T

Ein T-Objekt oder eine (beliebige informationslose) Markierung, dass nichts da ist: Ein T oder nichts

Either = T_1 + T_2

Der Entweder-Oder-Typ: ein T_1 oder ein T_2

ADT: Typ-Polynome + Typ-Gleichungen

Typen als Lösung von Typ-Gleichungen definieren

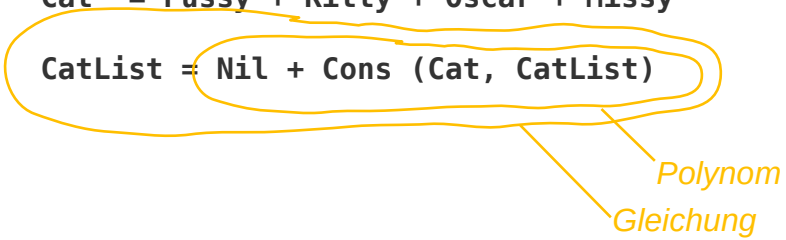
Datentypen können mit Hilfe von Rekursion / Summen und Produkten definiert werden
Damit wird die Definition von Typen möglich, deren Elemente

- **endlich**
- **aber unbeschränkt in ihrer Größe**

sind

Beispiel: Liste von Katzen

```
Pussy = Unit
Kitty = Unit
Oscar = Unit
Missy = Unit

Cat  = Pussy + Kitty + Oscar + Missy

CatList = Nil + Cons (Cat, CatList)
```



Die parameterlosen Konstruktoren

`Pussy`, `Kitty` `Oscar` ***und*** `Missy`

erzeugen (aus dem Nichts) Konstanten vom Typ `Cat`.

Der parameterlose Konstruktor

`Nil` ***erzeugt ein Datum vom Typ*** `CatList`.

Ebenso der Konstruktor

`Cons`, ***der ein Datum vom Typ*** `Cat` ***und***
ein Datum vom Typ `CatList` ***als Parameter hat.***

Typ-Gleichungen / rekursiv definierte Typen

Katzen-Listen in C

```
enum Cat { Pussy, Kitty, Oscar, Missy } Cat;

typedef struct CatList {
      enum { isNil, isCons } kind;
      enum Cat cat;
      struct CatList * lst;
} CatList;
```

***** : Der Pointer ist notwendig, um eine unbeschränkte Verschachtelung zu ermöglichen.**

In C und anderen Sprachen zur Systemprogrammierung können Pointer explizit vom Programm manipuliert werden. In anderen Sprachen sind Pointer hinter den Kulissen und dem direkten Zugriff durch das Programm (den tapsigen Anwendungsprogrammierer) entzogen.

Katzen-Listen in Scala

```
sealed trait Cat

object Pussy extends Cat
object Kitty extends Cat
object Oscar extends Cat
object Missy extends Cat

sealed trait CatList
object Nil extends CatList
case class Cons(head: Cat, tail: CatList) extends CatList
```

Katzen-Listen in Haskell

```
data Cat = Pussy | Kitty | Oscar | Missy

data CatList = Nil | Cons Cat CatList
```

Typ-Gleichungen / rekursiv definierte Typen

ADTs als Enums in Scala 3

Neben der Codierung via Vererbung kann ab Scala 3 auch die Enum-Notation verwendet werden.

```
enum Cat {
  case Pussy
  case Kitty
  case Oscar
  case Missy
}

enum CatList {
  case Nil
  case Cons(head: Cat, tail: CatList)
}
```

Enums sind ein rein syntaktisches Feature, das vom Compiler in Scala-Konstrukte umgesetzt wird.

Beispiel arithmetische Ausdrücke

Arithmetische Ausdrücke als ADT

```scala
sealed trait Exp
case class Const(v: Int) extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Mult(e1: Exp, e2: Exp) extends Exp
```

```scala
enum Exp {
  case Const(v: Int)
  case Add(e1: Exp, e2: Exp)
  case Mult(e1: Exp, e2: Exp)
}
```

Und natürlich auch generisch:

```scala
sealed trait Exp[+A]
case class Const(v: Int) extends Exp[A]
case class Add(e1: Exp, e2: Exp) extends Exp[A]
case class Mult(e1: Exp, e2: Exp) extends Exp[A]
```

```scala
enum Exp[+A] {
  case Const(v: A)
  case Add(e1: Exp[A], e2: Exp[A])
  case Mult(e1: Exp[A], e2: Exp[A])
}
```

GADTs : verallgemeinerte algebraische Datentypen

GADT = G*eneralized Algebraic Data Type*

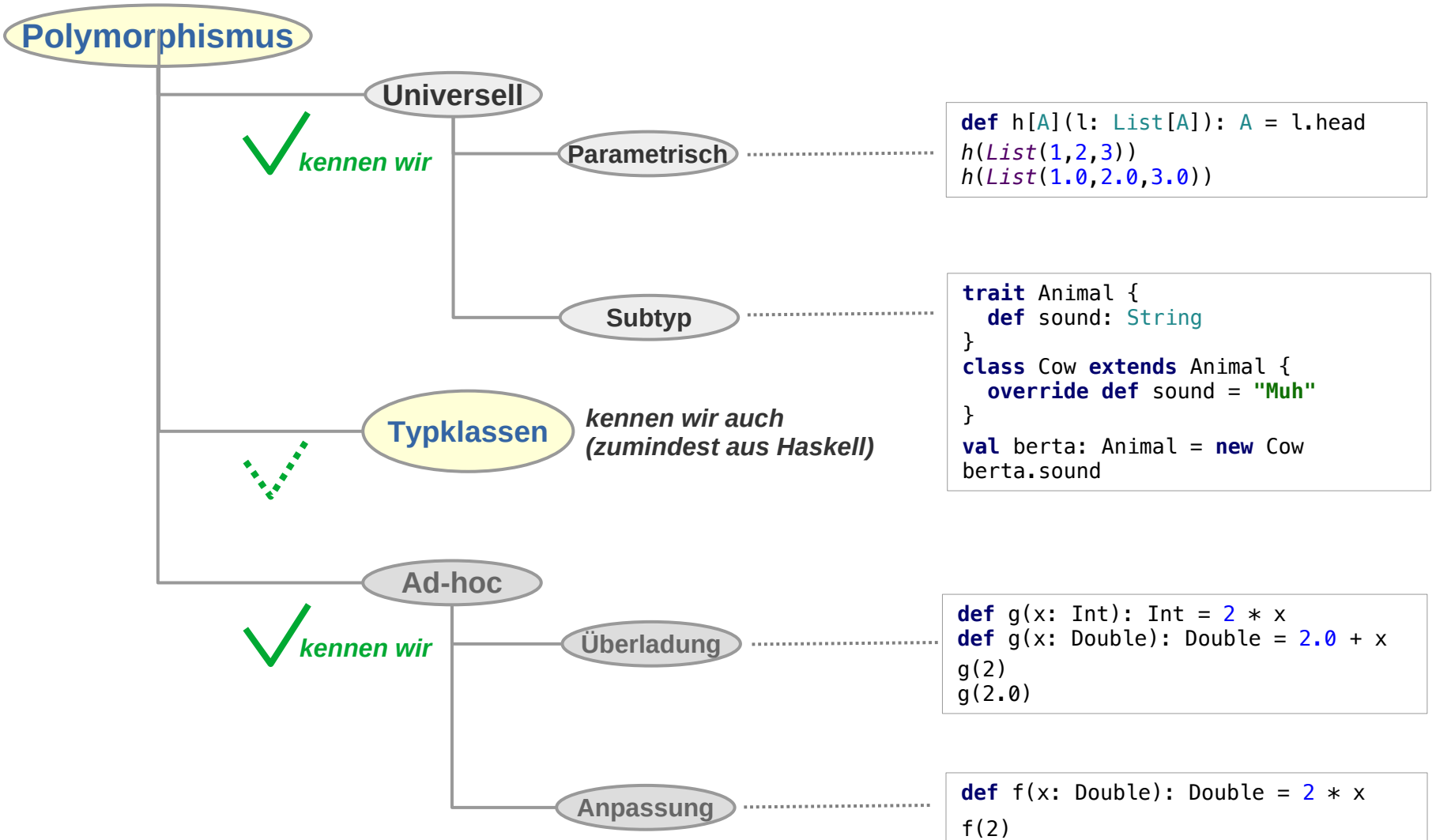
ADTs mit Varianten mit unterschiedlichen Typargumenten

Beispiel:

```
enum Exp[A] {
  case IntConst(i: Int)                       extends Exp[Int]
  case BoolConst(b: Boolean)                  extends Exp[Boolean]
  case Add(e1: Exp[Int], e2: Exp[Int])        extends Exp[Int]
  case Or(e1: Exp[Boolean], e2: Exp[Boolean]) extends Exp[Boolean]
}

import Exp._

def eval[A](e: Exp[A]): A = e match {
  case IntConst(i)  => i
  case BoolConst(b) => b
  case Add(e1: Exp[Int], e2: Exp[Int]) => eval(e1) + eval(e2)
  case Or(e1: Exp[Boolean], e2: Exp[Boolean]) => eval(e1) || eval(e2)
}

val eI = Add(Add(IntConst(1), IntConst(1)), IntConst(1))
val eB = Or(Or(BoolConst(true), BoolConst(true)), BoolConst(false))

val vI = eval(eI)  // 3
val vB = eval(eB)  // true
```

Typklassen:
Typen als (funktionale) Verhaltensmuster

Polymorphismus Übersicht:




```
def h[A](l: List[A]): A = l.head
h(List(1,2,3))
h(List(1.0,2.0,3.0))
```

```
trait Animal {
  def sound: String
}
class Cow extends Animal {
  override def sound = "Muh"
}
val berta: Animal = new Cow
berta.sound
```

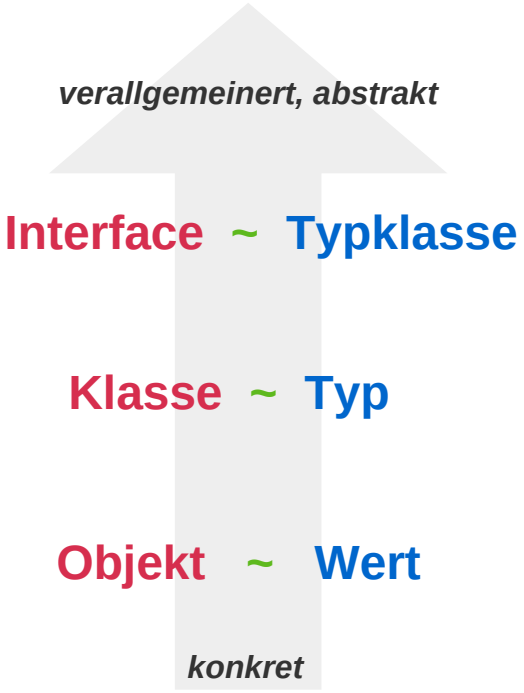
```
def g(x: Int): Int = 2 * x
def g(x: Double): Double = 2.0 + x
g(2)
g(2.0)
```

```
def f(x: Double): Double = 2 * x
f(2)
f(2.0)
```

Typklasse

Klassifikation von Typen nach unterstützten Funktionen

Funktionales Äquivalent zu Interfaces in der objektorientierten Programmierung



Typklasse

Klassifikation von Typen nach unterstützten Funktionen

Eine Typklasse

- definiert eine Klasse (= Kollektion) von Typen
- durch die Angabe von „Fähigkeiten“ die
- Werte haben müssen, die zu einem Typ der Klasse gehören
- Typklassen katalogisieren Typen entsprechend ihrer „Fähigkeiten“, sie sind darum mit Interfaces vergleichbar

```
class Eq a where
   (==) :: a → a → Bool
   (!=) :: a → a → Bool
```

Typklasse in Haskell

```
instance Eq Nat where
   x == y = natEq x y
```

Typklasseninstanz in Haskell

Beispiele

- *Geordnet*: Die Typen deren Werte mit „<“ verglichen werden können
- Mit Gleichheit: Typen, deren Werte auf Gleichheit getestet werden können
- ist ein *Monoid*: Typen die eine binäre assoziative Operation mit neutralem Element haben
- ist ein *Verband*: Geordnete Typen mit einem größten und einem kleinsten Element, und
- zwei assoziative Operationen Vereinigung und Schnitt (Mengen sind Verbände)
- ...

Typklassen: die Essenz

Mit Typklassen kann folgendes ausgedrückt werden, Beispiel Monoid:

- **Eine Signatur über einer ungebundenen Typ-Variablen**

  Eine Signatur ist eine Beschreibung einer Kollektion von Funktionen mit ihren Argument- und Ergebnistypen

```
class Monoid m where
    combine :: m -> m -> m
    unit :: m
```

Definition einer Typklasse: Signatur angeben

- **Eine Interpretation eines konkreten Typs in dieser Signatur**

  Ein realer Typ wird als Mitglied (Instanz) der Typklasse beschrieben indem
  - **der Typ der Typ-Variablen und**
  - **die Operationen des Typs denen der Signatur**

  zugeordnet werden

```
instance Monoid Integer where
    combine = (+)
    unit = 0
```

Instanziierung: Typvariable m und Operatoren interpretieren

Typklassen in Scala

Mit Typklassen kann folgendes ausgedrückt werden:

- **Eine Signatur über einer ungebundenen Typ-Variablen**
- **Eine Interpretation eines konkreten Typs in dieser Signatur**

Beispiel Monoid:

```haskell
class Monoid m where
    combine :: m -> m -> m
    unit :: m
```

```haskell
instance Monoid Integer where
    combine = (+)
    unit = 0
```

Haskell: Typklasse Monoid und Integer mit 0 und + als Monoid

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

```scala
given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Scala: Typklasse Monoid und Integer mit 0 und + als Monoid

Typklassen in Scala

Beispiel Monoid mit Anwendung

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

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(summon[Monoid[M]].unit)( (acc, a) =>
    summon[Monoid[M]].combine(acc, a))


given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}

val l = List(1,2,3)
val s = sumList(l)
```

Typklasse

Verwendung

Definition einer Instanz

*„**Summon** [...] mean to demand the presence of. Imply the exercise of authority.“* **Merriam-Webster**

Typklassen in Scala

Elimination von summon im Anwendungscode

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

object Monoid {
  def apply[M:Monoid] = summon[Monoid[M]]
}
```

oder

```
object Monoid {
  def apply[M](using m: Monoid[M]) = m
}
```

```
def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    Monoid[M].combine(acc, a))
```

Typklassen in Scala: „Syntax“

Syntax-Varianten: Funktions-Notation vs OO- / Extention- / Infix-Notation

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

object Monoid {
  def apply[M:Monoid] = summon[Monoid[M]]
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    Monoid[M].combine(acc, a))


given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}

val l = List(1,2,3)
val s = sumList(l)
```

Funktions-Notation

```scala
trait Monoid[M] {
  extension (x: M) def combine (y: M): M
  def unit: M
}

object Monoid {
  def apply[M](using m: Monoid[M]) = m
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    acc.combine(a))


given Monoid[Int] with {
  extension (x: Int) def combine (y: Int): Int = x + y
  def unit: Int = 0
}

val l = List(1,2,3)
val s = sumList(l)
```

OO- / Extension- / Infix-Notation

Typklassen in Scala: „Syntax“

Syntax-Varianten: Funktions-Notation vs OO / Extention-/ Infix-Notation

Operatoren können auch verwendet werden

```scala
trait Monoid[M] {
  extension (x: M) def ° (y: M): M
  def unit: M
}

object Monoid {
  def apply[M: Monoid] = summon[Monoid[M]]
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    acc ° a)


given Monoid[List[Int]] with {
  extension (x: List[Int]) def ° (y: List[Int]): List[Int] = x ::: y
  def unit: List[Int] = Nil
}

val l = List(List(1,2,3), List(4,5,6))
val s = sumList(l)
```

Typklassen für Typkonstruktoren

Typkonstruktoren (*higher kinded types*) K können einer Typklasse zugeordnet werden,
Damit werden Anforderungen formuliert, die

- **alle Typen K[A],**
  die mit dem Typkonstruktor K erzeugt werden können,

erfüllen müssen.

Beispiel Swapable: Alle Strukturen die „geswapt“ werden können

```
trait Swapable[S[_,_]] {
  def swap[A,B](s: S[A, B]): S[B, A]
}

object Swapable {
  def apply[F[_, _]: Swapable] = summon[Swapable[F]]
}

def swapAll[S[_, _]: Swapable, A, B](lst: List[S[A, B]]): List[S[B, A]] =
  lst.map(Swapable[S].swap(_))

case class Pair[A, B](a: A, b: B)

given Swapable[Pair] with {
  def swap[A, B](p: Pair[A,B]): Pair[B, A] = p match {
    case Pair(a, b) => Pair(b,a)
  }
}

val lst_1: List[Pair[Int, String]] = List(Pair(1, "a"), Pair(2, "b"))
val lst_2 = swapAll(lst_1) // List(Pair("a",1), Pair("b",2))
```

Typklassen für Typkonstruktoren

Beispiel Flatable

```scala
trait Flatable[F[_]] {
  def flatten[A](fa: F[A]): List[A]
}

object Flatable {
  def apply[F[_]: Flatable] = summon[Flatable[F]]
}

def flatList[F[_]: Flatable, A](lst: List[F[A]]): List[A] =
  lst.map(Flatable[F].flatten( _ )).flatten


case class Pair[A](x: A, y: A)

given Flatable[Pair] with {
  def flatten[A](p: Pair[A]): List[A] = p match {
    case Pair(a, b) => List(a,b)
  }
}

val lst_1: List[Pair[String]] = List(Pair("a", "b"), Pair("c", "d"))
val lst_2: List[String] = flatList(lst_1) // List("a", "b", "c", "d")
```

Typklassen für Typkonstruktoren

Beispiel Flatable / mit „Methoden-Syntax“

```scala
trait Flatable[F[_]] {
  extension[A] (fa: F[A]) def flatten: List[A]
}

object Flatable {
  def apply[F[_]: Flatable] = summon[Flatable[F]]
}

def flatList[F[_]: Flatable, A](lst: List[F[A]]): List[A] =
  lst.map( _.flatten ).flatten


case class Pair[A](x: A, y: A)

given Flatable[Pair] with {
  extension[A] (p: Pair[A])  def flatten : List[A] = p match {
    case Pair(a, b) => List(a,b)
  }
}

val lst_1: List[Pair[String]] = List(Pair("a", "b"), Pair("c", "d"))
val lst_2: List[String] = flatList(lst_1) // List("a", "b", "c", "d")
```

Typrelationen: Typklassen mit mehr als einem Typ-Parameter

Beispiel Rechnen mit Einheiten (und Phantom-Typen)

Die Typen U1 und U2 können in einer Konversionsrelation stehen

Die Konversionsrelation ist eine Typklasse mit zwei (Typ-) Argumenten

```
final case class Quantity[U](value: Double)
```

Werte mit einer Einheit U. U ist ein „Phantom-Typ“
Quantity *enthält nichts vom Typ U*

```
trait Convertible[U1, U2] {
  def convert(u1: Double): Double
}
```

Konvertiere von U1 nach U2

```
def add[U1, U2](x: Quantity[U1], y: Quantity[U2])
               (using ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(ev.convert(x.value) + y.value)
```

Werte mit Einheit U1 und U2 können addiert werden, wenn es eine Konversion von U1 nach U2 gibt

```
trait Km
trait Mile
```

Zwei Einheiten mit ihren Konversionen

```
given Convertible[Km, Km] with {
  def convert(km: Double): Double = km
}

given Convertible[Mile, Mile] with {
  def convert(mile: Double): Double = mile
}

given Convertible[Mile, Km] with { // mile => km
  def convert(mile: Double): Double = mile * 1.60934
}

given Convertible[Km, Mile] with { // km => mile
  def convert(km: Double): Double = km / 1.60934
}
```

Typrelationen: Typklassen mit mehr als einem Typparameter

Beispiel Rechnen mit Einheiten

Anwendung:

```
val v1: Quantity[Km]   = Quantity[Km](2.0)
val v2: Quantity[Km]   = Quantity[Km](1.0)
val v3: Quantity[Mile] = Quantity[Mile](1.0)


val v1Plus2 = add(v1, v2) // Quantity(3.0)
val v1Plus3 = add(v1, v3) // Quantity(2.2427454732996135)
val v3Plus1 = add(v3, v1) // Quantity(3.60934)
val v3Plus3 = add(v3, v3) // Quantity(2.0)
```

Typrelationen: Typklassen mit mehr als einem Typparameter

Beispiel **Rechnen mit Einheiten / Noch etwas schöner mit Extension-Methods:**

```
final case class Quantity[U](value: Double)

trait Convertible[U1, U2] {
  def convert(u1: Double): Double
}

def add[U1, U2](x: Quantity[U1], y: Quantity[U2])
               (using ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(ev.convert(x.value) + y.value)

trait Km
trait Mile

import scala.language.postfixOps
```

```
object UnitOps {
  extension (x: Double)  def km : Quantity[Km] =  Quantity[Km](x)
  extension (x: Double)  def mile : Quantity[Mile] =  Quantity[Mile](x)
  extension[U1, U2] (x: Quantity[U1]) def + (y: Quantity[U2])(using ev: Convertible[U1, U2]) = add(x, y)
}
import UnitOps._
```

Typrelationen: Typklassen mit mehr als einem Typparameter

Beispiel Rechnen mit Einheiten / Noch etwas schöner mit Extension-Methods:

```
given Convertible[Km, Km] with {
  def convert(km: Double): Double = km
}
given Convertible[Mile, Mile] with {
  def convert(mile: Double): Double = mile
}
given Convertible[Mile, Km] with { // mile => km
  def convert(mile: Double): Double = mile * 1.60934
}
given Convertible[Km, Mile] with { // km => mile
  def convert(km: Double): Double = km / 1.60934
}

val v1 = 2.0 km
val v2 = 1.0 km
val v3 = 1.0 mile

val v1Plus2 = v1 + v2 // Quantity(3.0)
val v1Plus3 = v1 + v3 // Quantity(2.2427454732996135)
val v3Plus1 = v3 + v1 // Quantity(3.60934)
val v3Plus3 = v3 + v3 // Quantity(2.0)
```

Erweiterungen von Typklassen

Typklassen können in hierarchischen Beziehungen stehen

Beispiel:

- **Eine Halbgruppe hat eine zweistellige Operation**
- **Ein Monoid ist eine Halbgruppe mit einem neutralen Element**

Ein Monoid ist darum auch immer eine Halbgruppe

Diese Beziehung kann auf zwei Arten in Scala dargestellt werden:

- **als OO-Vererbung**
- **als „Typklassenvererbung“: also als Typklassen-Anforderung an den generischen Parameter**

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

OO-Vererbung

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A: Semigroup]  {
  def empty: A
}
```

Typanforderung an den generischen Parameter

Typklassen und „Typklassen-Vererbung“

OO-Vererbung

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}

given Semigroup[Int] with {
  def combine(x: Int, y: Int): Int = x+y
}

given Monoid[Int] with {
  def empty: Int = 0
  def combine(x: Int, y: Int): Int = x+y // muss definiert werden
}

def reduceSemiGroup[A: Semigroup](lst: List[A]): Option[A] = lst match {
  case Nil => None
  case head :: Nil => Some(head)
  case head :: tail =>
    reduceSemiGroup(tail).map(summon[Semigroup[A]].combine(head,_))
}

def reduceMonoid[A: Monoid](lst: List[A]): A = lst match {
  case Nil => summon[Monoid[A]].empty
  case _ => reduceSemiGroup(lst).get  // Monoid ist auch Semigroup
}

val lst: List[Int] = List(1,2,3)
val sum1 = reduceSemiGroup(lst)
val sum2 = reduceMonoid(lst)
```

Typklassen und „Typklassen-Vererbung“

Anforderungs-Vererbung

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A: Semigroup]  {
  def empty: A
}

given Semigroup[Int] with {
  def combine(x: Int, y: Int): Int = x+y
}

given Monoid[Int] with {
  def empty: Int = 0
  //nicht notwendig: def combine(x: Int, y: Int): Int = x+y
}

def reduceSemiGroup[A: Semigroup](lst: List[A]): Option[A] = lst match {
  case Nil => None
  case head :: Nil => Some(head)
  case head :: tail =>
    reduceSemiGroup(tail).map(summon[Semigroup[A]].combine(head,_))
}

def reduceMonoid[A: Monoid: Semigroup](lst: List[A]): A = lst match {
  case Nil => summon[Monoid[A]].empty
  case _ =>
    reduceSemiGroup(lst).get
}

val lst: List[Int] = List(1,2,3)
val sum1 = reduceSemiGroup(lst)
val sum2 = reduceMonoid(lst)
```