# Funktionale Programmierung

## Sommersemester 2025

Prof. Dr. Jakob Rehof
M. Sc. Felix Laarmann
TU Dortmund
LS XIV Software Engineering

# Diese Vorlesung:

Hintergründe zu

- Homomorphismen
- Termalgebren und Konstruktoren
- Wirkung und Zustand

# Homomorphismus

# Homomorphismus

*Homomorphismus* heißt in der Mathematik:

- eine *strukturtreue* (*strukturbewahrende*) Abbildung

Aus dem Altgriechischen:

- *homós* = gleich (ähnlich)
- *morphé* = Form

# Homomorphismus

Seien $\langle \mathbf{A}, \circ \rangle$ und $\langle \mathbf{B}, \bullet \rangle$ zwei algebraische Strukturen mit Operationen (jeweils) $\circ$ und $\bullet$, wobei

$$\circ : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$$

$$\bullet : \mathbf{B} \times \mathbf{B} \to \mathbf{B}$$

Eine Abbildung $\varphi : \mathbf{A} \to \mathbf{B}$ ist dann ein Homomorphismus (von $\mathbf{A}$ nach $\mathbf{B}$), wenn für alle $x, y \in \mathbf{A}$ gilt:

$$\varphi(x \circ y) = \varphi(x) \bullet \varphi(y)$$

Weitere algebraische Struktur wird automatisch erhalten in $\varphi(\mathbf{A}) \subseteq \mathbf{B}$, Beispiel Neutralelement:

$$\varphi(x) = \varphi(1 \circ x) = \varphi(1) \bullet \varphi(x)$$

Also, $\varphi(1) \bullet \varphi(x) = \varphi(x)$.

# Homomorphismus

Bekannte Homomorphismen

- Exponentialfunktionen. Mit $\varphi(x) = e^x$ haben wir $\varphi(x + y) = \varphi(x) \cdot \varphi(y)$
- Lineare Abbildungen sind Homomorphismen zwischen Vektorräumen: $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$, $af(\mathbf{x}) = f(a\mathbf{x})$
- Substitutionen sind Homomorphismen in Termalgebren: $S(\mathbf{t}(x, y)) = \mathbf{t}(S(x), S(y))$
- Graphhomomorphismus $f : \langle V_G, E_G \rangle \to \langle V_H, E_H \rangle$: $(u, v) \in E_G \Rightarrow (f(u), f(v)) \in E_H$

Das Prinzip auf Datenstrukturen übertragen, zum Beispiel:

- Auf Paaren hätten wir danach $f^*(x, y) = (f(x), f(y))$
- Auf Listen hätten wir danach $f^*(x : xs) = f(x) \bullet f^*(xs)$

# Homomorphismus

Setzen wir $(\varphi \times \varphi)(x, y) = (\varphi(x), \varphi(y))$, können wie auch die Homomorphismus-eigenschaft durch die *Kommutatvität des Diagramms ausdrücken:*

$$
\begin{tikzcd}
A \times A \arrow[r, "\varphi \times \varphi"] \arrow[d, swap, "\circ"] & B \times B \arrow[d, "\bullet"] \\
A \arrow[r, "\varphi"] & B
\end{tikzcd}
$$

# Termalgebren und Konstruktoren

# Termalgebren und Konstruktoren

- Termalgebra
  - Trägermenge: **Syntaxbäume**
  - Operatoren: **Konstruktoren**
- Intuitiv kann man *Konstruktoren* als abstrakte Funktionssymbole auffassen, die keine weiteren Berechnungsregeln haben als die Formierung von syntaktischen Ausdrücken
- Beispiel: Anwendung von `f` auf *e*, geschrieben „`f(`*e*`)`“, ergibt den Syntaxbaum

$$\begin{array}{c} \mathtt{f} \\ | \\ e \end{array}$$

# Termalgebren und Konstruktoren

- Beispiel Grammatik (BNF):

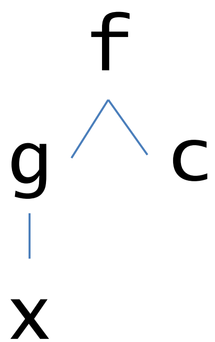
$$E ::= X \mid c \mid \mathtt{g}(E) \mid \mathtt{f}(E, E)$$

- Bestandteile: Variablen *X*, Konstanten *c*, Funktionssymbol `g` mit Stelligkeit 1, Funktionssymbol `f` mit Stelligkeit 2
- Wir fassen Konstanten als Funktionssymbole mit Stelligkeit 0 auf und schreiben einfach „`c`" für „`c()`"
- Konstanten und Funktionssymbole werden als *Konstruktoren* bezeichnet
- PS: „Stelligkeit" heißt auf Englisch „arity" - „nullary", „unary" „binary", „ternary", …

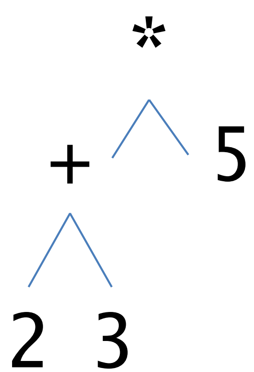
# Termalgebren und Konstruktoren

- Konkrete Syntax: Strings
- Abstrakte Syntax: Bäume

„f(g(x),c)“




„(2+3)*5“



# Termalgebren und Konstruktoren

- Beispiel **Typkonstruktoren**
  - Syntax

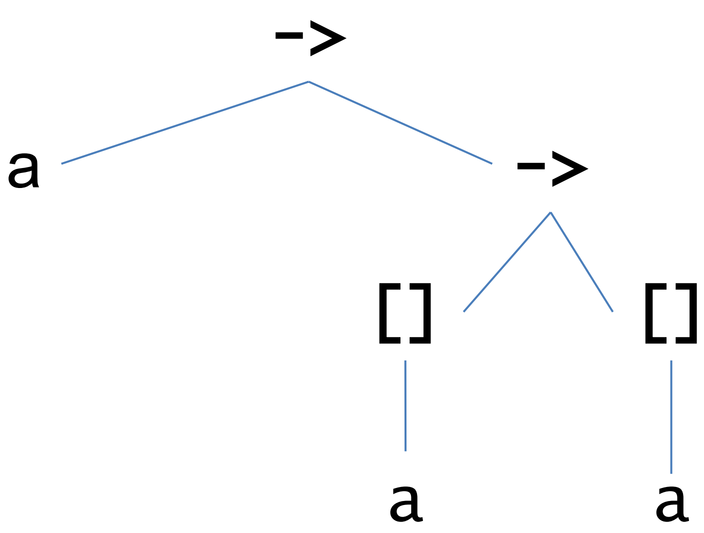
    `T` ::= *tv* | *tc* | | `[T]` | `(T,T)` | `T -> T`
  - Typvariablen *tv*: a, b, c, …
  - Typkonstruktoren *tc*:

| | |
|---|---|
| `String`, `Bool`, `Char`, `Int`, … | |
| | Konstanten (Konstruktoren mit Stelligkeit 0) |
| `[_]` | Konstruktor mit Stelligkeit 1 |
| `(_,_)` | Konstruktor mit Stelligkeit 2 |
| `_ -> _` | Konstruktor mit Stelligkeit 2 |

# Termalgebren und Konstruktoren

- Beispiel:

Der Typ `a -> [a] -> [a]` ist „syntactic sugar“ (Schreibweise) für `a -> ([a] -> [a])` und ist eine Notation für den Syntaxbaum



# Termalgebren und Konstruktoren

- Beispiel **Datenkonstruktoren**
  - Syntax

    D ::= *v* | *c* | | D : D | (D,D) | …

  - Variablen: x, y, z, …
  - Konstruktoren:

    [], 0, 1, True, False, …

    Konstanten (Konstruktoren mit Stelligkeit 0)

    _ : _ Konstruktor mit Stelligkeit 2

    (_,_) Konstruktor mit Stelligkeit 2

# Termalgebren und Konstruktoren

- Beispiel
  - Syntax

$$D ::= v \mid c \mid \mid D : D \mid (D,D) \mid \ldots$$

$v$ = `x`, `y`, `z`, …, $c$ = `[]`, `0`, `1`, `True`, `False`, …

Der Ausdruck `[0, 1, 2, 3, 4]` ist „syntactic sugar“ für den Syntaxbaum, welcher durch

```
0 : (1 : (2 : (3 : (4 : []))))
```

definiert wird.

# Termalgebren und Konstruktoren

- *Substitutionen sind Homomorphismen in Termalgebren*
- Sei S = {X1 $\mapsto$ t1, …, Xn $\mapsto$ tn}. Dann bestimmt S eindeutig den Homomorphismus S*, gegeben durch
  - S*(X) = S(X) falls X $\in$ {X1, …, Xn}, ansonsten S*(X) = X
  - S*(c) = c
  - S*(f(t1, …, tn)) = f(S*(t1), …, S*(tn))
- Weil S* eindeutig von S bestimmt ist, benutzt man für S* gern auch den Namen S

# Wirkung und Zustand

Was ist eine Funktion, mathematisch gesehen?

Seien $A$ und $B$ Mengen. Eine (totale) Funktion $f : A \to B$ ist eine Relation $f \subseteq A \times B$

$$f = \{(x, y) \in A \times B \mid x \in A, y \in B\}$$

mit den Eigenschaften

- *Totalität*: $\forall x \in A.\ \exists y \in B.\ (x, y) \in f$
- *Funktionalität*: $\forall x_1, x_2 \in A.\ \forall y_1, y_2 \in B.$
  $[(x_1, y_1) \in f \wedge (x_2, y_2) \in f \wedge x_1 = x_2] \Rightarrow y_1 = y_2$

und für $(x, y) \in f$ schreiben wir $y = f(x)$.

PS: Nicht zu verwechseln mit den weiteren Eigenschaften, die Funktionen haben können aber nicht haben müssen:

- *Injektivität*: $\forall x_1, x_2 \in A.\ f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
- *Surjektivität*: $\forall y \in B.\ \exists x \in A.\ y = f(x)$

## Wirkung und Zustand

In einer *imperativen* Sprache kann ich schreiben:

```
int count := 0;

int f(int x) = {count := count + 1; return x + count}
```

wobei `count` eine globale Variable ist.
Das macht allerdings keine Funktion aus `f`! Denn ich kriege bei jedem Aufruf von `f` mit demselben Argument ein Unterschiedliches “Abbild”:

```
y1 := f(5)
y2 := f(5)
```

Hier gilt mit Sicherheit `y1` $\neq$ `y2`. Also ist die Funktionalitätseigenschaft *nicht* erfüllt.

Dies liegt an der Verwendung von Operationen, die *Wirkungen* (*side effects*) auf einen globalen *Zustand* (hier durch `count` implementiert) haben.

# Referentielle Transparenz

Eine Konsequenz der Funktionalitätseigenschaft ist die sogenannte **referentielle Transparenz**:

- *Ein Ausdruck ist mit seinem Wert in allen Kontexten austauschbar*

Daher, ob ich in einer *funktionalen* Sprache zum Beispiel schreibe

```
let y = f(5)
in
    (y, y)
```

oder ich schreibe

```
(f(5), f(5))
```

bleibt das Ergebnis identisch.

# Referentielle Transparenz

Eine Konsequenz der Funktionalitätseigenschaft ist die sogenannte **referentielle Transparenz**:

- *Ein Ausdruck ist mit seinem Wert in allen Kontexten austauschbar*

Aber *nicht* so in der imperativen Version:

```
let y = f(5)
in
    (y, y)
```

ergibt den Wert `(5, 5)`, wohingegen

```
(f(5), f(5))
```

den Wert `(5, 6)` ergibt.

# Kontrolloperationen (control operators, exceptions)

*Kontrolloperationen* können als Operationen verstanden werden, welche den *Evaluierungskontext* manipulieren. Kontexte können wir als Ausdrücke mit einem “Loch” formalisieren. Beispiel:

$$E ::= [] \mid n \mid E + E \mid E * E$$

Hier bezeichnet [] ein Loch in einem Ausdruck (dem Kontext), das mit einem Ausdruck $M$ gefüllt werden kann, und das Ergebnis schreiben wir $E[M]$. Zum Beispiel sei $E = 3 + []$ und $M = 2$, dann ist $E[M] = 3 + 2$.

Wir können nun das Prinzip der referentiellen Transparenz mit dem Kontextbegriff formalisieren:

Für alle Kontexte $E$, alle Ausdrücke $M$ und Werte $W$:

$$M = W \quad \Rightarrow \quad E[M] = E[W]$$

# Kontrolloperationen (control operators, exceptions)

Mittels Kontexte $E$ können wir zum Beispiel eine Kontrolloperation $\mathcal{A}$ definieren:

$$E[\mathcal{A}(M)] = M$$

für beliebige Kontexte $E$. Diese Operation "schmeißt" das Argument $M$ als Ergebnis der Berechnung über jeden Kontext hinweg. Das wäre eine "abort" oder "throw" Operation.

Wir haben (im leeren Kontext $[]$)

$$\mathcal{A}(0) = [][\mathcal{A}(0)] = [\mathcal{A}(0)] = 0$$

Also wäre 0 der "Wert" von $\mathcal{A}(0)$. Ersetzen wir mit diesem Wert im Kontext $E = 3 + []$, bekommen wir also $E[0] = 3 + 0 = 3$. Ersetzen wir aber mit dem Ausdruck (also $\mathcal{A}(0)$)bekommen wir

$$E[\mathcal{A}(0)] = 3 + \mathcal{A}(0) = 0$$

Kontrolloperationen sind also *nicht* referentiell transparent.

# Store transformation

Man kann in einer funktionalen Sprache einen globalen Zustand simulieren, indem man *alle* Funktionen mit einem extra Argument ausstattet, welches den globalen Zustand repräsentiert.

Eine imperative “Funktion” $f : A \to B$ wird durch eine eigentliche Funktion $F : (A, Z) \to (B, Z)$ ersetzt, wobei $Z$ der Typ des globalen Zustands ist. Den Übergang von $f$ zu $F$ nennt man manchmal *store transformation*. Man muss dafür sorgen, daß die “Wirkungen” von $f$ auf den “Zustand” in $F$ immer zurück gegeben wird und im Programm korrekt weiter gereicht wird (man nennt dies manchmal *store threading*).

Zum Beispiel:

```
F (x,count) = let count' = count+1 in (x+count',count')
```

mit Verwendung

```
let (y1,c1) = F (5,0)
    (y2,c2) = F (5,c1)
```

Die Funktion $F$ ist eine richtige Funktion, mit referentieller Transparenz und allem.

## So, what's the problem?

Das Problem bei *store transformation* ist allerdings, daß das *gesamte* Programm, mit dem totalen Programmkontext, transformiert werden muss. Der globale Zustand muss eben durch eine globale Transformation simuliert werden. Dies betrifft auch mögliche Bestandteile des Programms, die gar nicht vom Zustand abhängen!

## The best of both worlds?

Könnten wir ein Konzept finden, mit welchem wir die Wirkungen auf den Zustand so *kappseln* könnten, dass nur die Bereiche des Programms, die wirklich vom Zustand abhängen, betroffen wären? Man spricht manchmal von *state encapsulation*.

## Enter *monads*!

## Kategorie $\mathcal{C}$

Objekte $ob(\mathcal{C})$

$$A \qquad B \qquad C \qquad D \qquad \ldots$$

Morphismen $hom(\mathcal{C})$

$$A \qquad B \xrightarrow{f} C \qquad D \qquad \ldots$$

Komposition und Identität

$$\begin{tikzcd} A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[rr, bend right, dashed, "g \circ f"'] & B \arrow[r, "g"] & C & D & \ldots \end{tikzcd}$$

Eine Kategorie $\mathcal{C}$ besteht aus:

- Eine Klasse $ob(\mathcal{C})$ von *Objekten* (objects)
- Eine Klasse $hom(\mathcal{C})$ von *Morphismen* (morphisms). Ein Morphismus $f$ hat ein Objekt $dom(f)$ als Quelle (domain) und ein Objekt $cod(f)$ als Ziel (codomain). Wir schreiben z.B. einen Morphismus $f$ mit $dom(f) = A$ und $cod(f) = B$ als $f : A \to B$ ("$f$ ist ein Morphismus von $A$ nach $B$). Die Klasse solcher Morphismen wird auch mit $hom(A, B)$ oder $hom_{\mathcal{C}}(A, B)$ benannt.
- Eine binäre Abbildung $\circ$, genannt *Komposition von Morphismen*, wobei $\circ : hom(B, C) \times hom(A, B) \to hom(A, C)$ (für $f : A \to B$ und $g : B \to C$ haben wir $g \circ f : A \to C$), mit folgenden Eigenschaften:
  - *Assoziativität.* Für alle $f : A \to B$, $g : B \to C$, $h : C \to D$ gilt $h \circ (g \circ f) = (h \circ g) \circ f$
  - *Identität.* Für jedes Objekt $A$ existiert ein Morphismus $1_A$ (genannt *Identität für* $A$), wobei für alle $f : A \to B$ gilt $1_B \circ f = f = f \circ 1_A$

# Beispiele

- Die Kategorie **Set**
  - $ob(\mathbf{Set})$ = Klasse von Mengen
  - $hom(A, B) = B^A$, alle Funktionen von $A$ nach $B$
- Die Kategorie $\mathbf{Pos}(P)$, $P$ eine partielle Ordnung $\langle P, \leq_P \rangle$
  - $ob(\mathbf{Pos}(P))$ = die Elemente von $P$
  - $hom(x, y) = \{(x, y)\}$, wenn $x \leq y$, ansonsten $\emptyset$
- Die Kategorie **PoSet**
  - $ob(\mathbf{PoSet})$ = partiell geordnete Mengen
  - $hom(P, Q)$ = monotone Funtionen von $P$ nach $Q$
- **Top**, die Kategorie der topologischen Räume (Objekte) und stetigen Abbildungen (Morphismen). Eine Unterkategorie ist beispielsweise die volle Unterkategorie **KHaus** der kompakten Hausdorff-Räume.
- Die Kategorie $\mathbf{Mon}(M)$, $\langle M, \bullet \rangle$ ein Monoid
  - $ob(\mathbf{Mon}(M)) = \{M\}$
  - $hom(M, M)$ = alle Abbildungen $f_m : M \to M$ mit $f_m(x) = m \bullet x$, $x \in M$

# Funktor

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* $F$ von $\mathcal{C}$ nach $\mathcal{D}$ ist eine Abbildung mit den Eigenschaften

- $F$ bildet jedes Objekt $A \in ob(\mathcal{C})$ in ein Objekt $F(A) \in ob(\mathcal{D})$ ab
- $F$ bildet jeden Morphismus $f : A \to B \in hom(\mathcal{C})$ in einen Morphismus $F(f) : F(A) \to F(B) \in hom(\mathcal{D})$ ab, wobei folgende Bedingungen gelten:
  - $F(1_A) = 1_{F(A)}$ für alle $A \in ob(\mathcal{C})$
  - $F(g \circ f) = F(g) \circ F(f)$ für alle $f : A \to B, g : B \to C \in hom(\mathcal{C})$

$$\begin{tikzcd}
\mathcal{C} & A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
\mathcal{D} & F(A) \arrow[r, "F(f)"] & F(B)
\end{tikzcd}$$

# Funktor

$$F(1_A) = 1_{F(A)}$$

$$\begin{tikzcd}
A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
F(A) \arrow[loop left, "1_{F(A)}"] \arrow[r, "F(f)"] & F(B)
\end{tikzcd}$$

# Funktor

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F"'] \arrow[rr, bend left, dashed, "g \circ f"] & B \arrow[r, "g"] \arrow[d, "F"] & C \arrow[d, "F"] \\
F(A) \arrow[r, "F(f)"] \arrow[rr, bend right, dashed, "F(g) \circ F(f)"'] & F(B) \arrow[r, "F(g)"] & F(C)
\end{tikzcd}
$$

# Funktor

$$
\begin{tikzcd}[column sep=huge, row sep=large]
A \arrow[r, "f"] \arrow[d, "F"'] \arrow[rr, bend left, dashed, "g \circ f"] & B \arrow[r, "g"] \arrow[d, "F"] \arrow[dr, phantom, "F"] & C \arrow[d, "F"] \\
F(A) \arrow[r, "F(f)"] \arrow[rr, bend right, dashed] & F(B) \arrow[r, "F(g)"] & F(C)
\end{tikzcd}
$$

$$F(g) \circ F(f) = F(g \circ f)$$

Funktoren sind Homomorphismen zwischen Kategorien

# Haskell Funktor

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
F(A) \arrow[r, "F(f)"] & F(B)
\end{tikzcd}
$$

# Haskell Funktor

- Objekte $\mapsto$ Typen
- Morphismen $\mapsto$ Funktionen
- Funktoren $\mapsto$ Typkonstruktoren $F :: * \to *$ mit Typklassen

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
F(A) \arrow[r, "F.\mathtt{fmap}(f)"] & F(B)
\end{tikzcd}
$$

# Haskell Funktor

! In diesen Folien schreibe ich gerne Objekte mit großen Buchstaben (wie A, B,…) . Wenn wir Haskell-Typen als Objekte betrachten, ergibt dies eine Notationskonflikt, weil Typen in Haskell mit klein geschrieben werden. Lassen Sie sich bitte nicht dadurch verwirren !

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\ F(A) \arrow[r, "F(f)"] & F(B) \end{tikzcd}$$

# Haskell Funktor

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

- $\texttt{fmap id} = \texttt{id}$
- $\texttt{fmap (g . f)} = \texttt{(fmap g) . (fmap f)}$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
F(A) \arrow[r, "F.\mathtt{fmap}(f)"] & F(B)
\end{tikzcd}$$

# Haskell Funktor

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

- fmap id = id
- fmap (g . f) = (fmap g) . (fmap f)

Jetzt wissen wir, woher diese semantischen Bedingungen kommen

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F"'] & B \arrow[d, "F"] \\
F(A) \arrow[r, "F.\mathtt{fmap}(f)"] & F(B)
\end{tikzcd}
$$