

# Funktionale Programmierung

## Sommersemester 2025

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

## Diese Vorlesung:

- Organisatorisches zu den Klausuren
- Übersicht über die behandelten Modulinhalte
- Kategorientheorie
- Ausblick

## Organisatorisches zu den Klausuren

- Die erste Klausur wird am Mittwoch, den **06.08.2025** von 15:30h-17:45h im **Audimax** stattfinden.
- Die zweite Klausur wird am Dienstag, den **16.09.2025** von 14:00h-16:15h im **Audimax** stattfinden.
- Die Dauer beträgt **120 Minuten**.
- Es sind keine zusätzlichen Hilfsmittel erlaubt.

## Infos zur Klausur

- Es werden Aufgaben ähnlich zu den Übungsaufgaben gestellt.
- Es wird keine Aufgabe zur IO-Monade geben.
- Ansonsten ist der gesamte behandelte Stoff relevant.
  
- Funktionen aus dem Haskell-Prelude dürfen verwendet werden, sowie die Funktion „guard“ aus dem Modul Control.Monad.
- Alle anderen Funktionen müssen selbst definiert werden, sofern man sie nutzen möchte.
- LanguageExtensions, andere Module etc. dürfen nicht zum Lösen der Aufgaben vorausgesetzt/verwendet werden.

# Haskell

Wir haben behandelt:

- Das Definieren von Funktionen mittels Rekursion, Pattern Matching, Guards, Faltungen, ...
- Die Listenmonade und Funktionen auf Listen, sowie Listenfaltungen, die Listenkomprehension als syntaktischen Zucker für ( $\gg=$ ), ...
- Weitere Datentypen aus dem Prelude, wie z.B. Maybe, Either, (,), etc. und die Definition eigener Datentypen, wie z.B. Nat, unterschiedliche Darstellungen von Bäumen, ...
- Typklassen wie Show, Eq, Ord, Functor, Applicative, Monad, etc. und die entsprechenden Instanziierungen.
- List-, Maybe-, Either-, Reader-, Writer- und State-Monaden
- ...

# Lambda-Kalkül

Wir haben behandelt:

- Das ungetypte Lambda-Kalkül:
  - Syntax von Termen (3 Konstruktoren!)
  - $\alpha$ -Äquivalenz
  - Substitution
  - $\beta$ -Reduktion
  - Unterschiedliche Reduktionsstrategien
  - ...
- Das einfach getypte Lambda-Kalkül:
  - Einfaches Typsystem (3 Regeln!)
  - Typinferenz & -check mit Ableitungsbäumen
  - ...
- Church-Kodierungen

# Klausur

Gibt es noch Fragen zur Klausur?

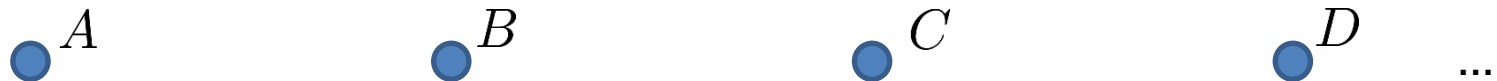
# Kategorientheorie

- Varianz
- Natürliche Transformation

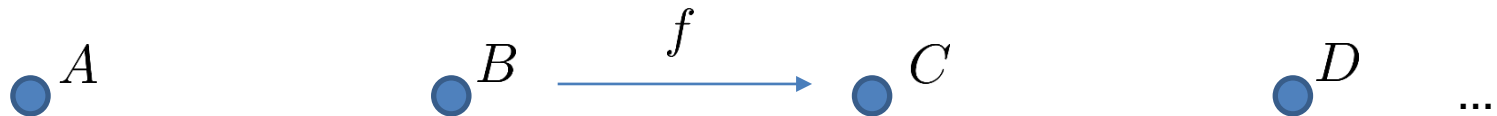


## Kategorie $\mathcal{C}$

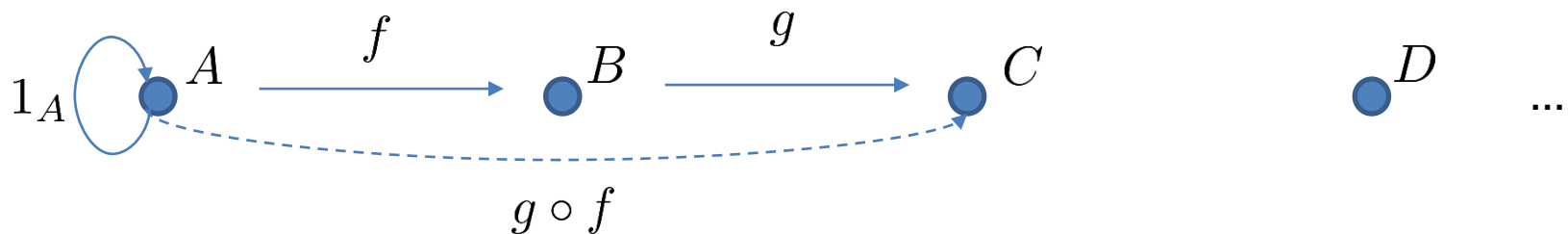
Objekte  $ob(\mathcal{C})$



Morphismen  $hom(\mathcal{C})$



Komposition und Identität



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  $f : A \rightarrow B$  (“ $f$  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  $\circ : hom(B, C) \times hom(A, B) \rightarrow hom(A, C)$  (für  $f : A \rightarrow B$  und  $g : B \rightarrow C$  haben wir  $g \circ f : A \rightarrow C$ ), mit folgenden Eigenschaften:
  - *Assoziativität*. Für alle  $f : A \rightarrow B$ ,  $g : B \rightarrow C$ ,  $h : C \rightarrow D$  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  $f : A \rightarrow B$  gilt  $1_B \circ f = f = f \circ 1_A$

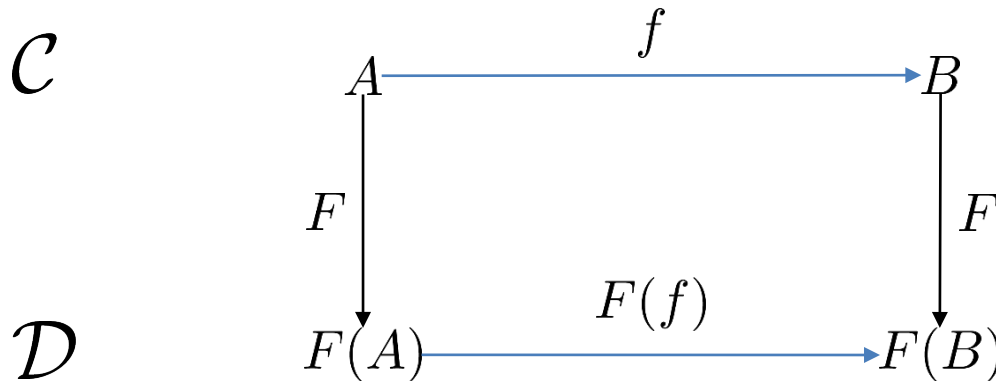
## Beispiele

- Die Kategorie **Set**
  - $ob(\mathbf{Set}) = \text{Klasse von Mengen}$
  - $hom(A, B) = B^A$ , alle Funktionen von  $A$  nach  $B$
- Die Kategorie **Pos**( $P$ ),  $P$  eine partielle Ordnung  $\langle P, \leq_P \rangle$ 
  - $ob(\mathbf{Pos}(P)) = \text{die Elemente von } P$
  - $hom(x, y) = \{(x, y)\}$ , wenn  $x \leq y$ , ansonsten  $\emptyset$
- Die Kategorie **PoSet**
  - $ob(\mathbf{PoSet}) = \text{partiell geordnete Mengen}$
  - $hom(P, Q) = \text{monotone Funktionen von } P \text{ nach } Q$
- **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 **Mon**( $M$ ),  $\langle M, \bullet \rangle$  ein Monoid
  - $ob(\mathbf{Mon}(M)) = \{M\}$
  - $hom(M, M) = \text{alle Abbildungen } f_m : M \rightarrow M \text{ mit } f_m(x) = m \bullet x, x \in M$

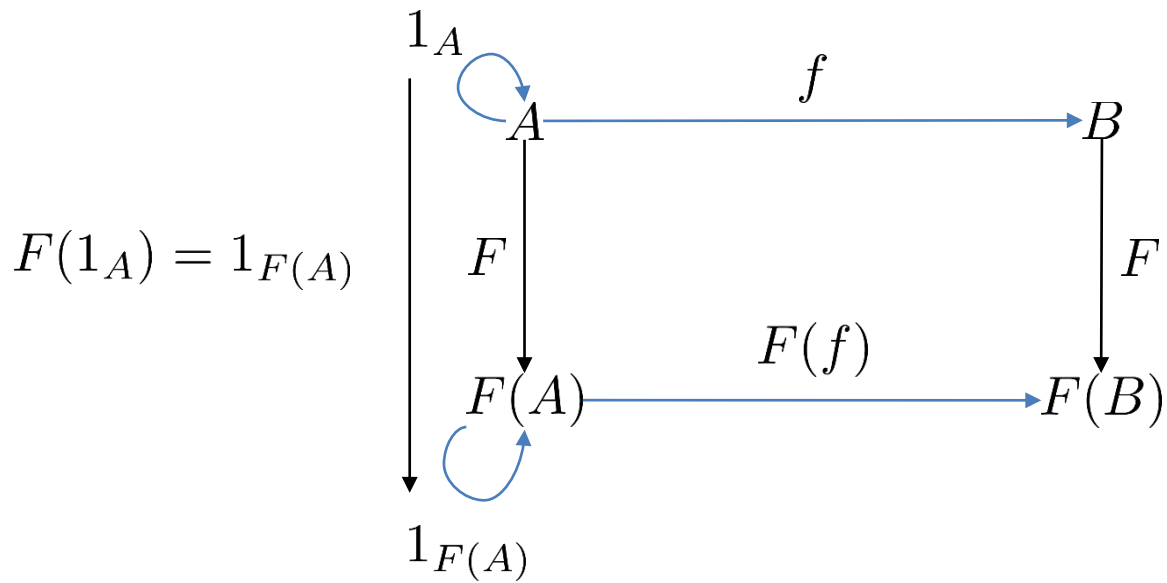
## Funktor

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

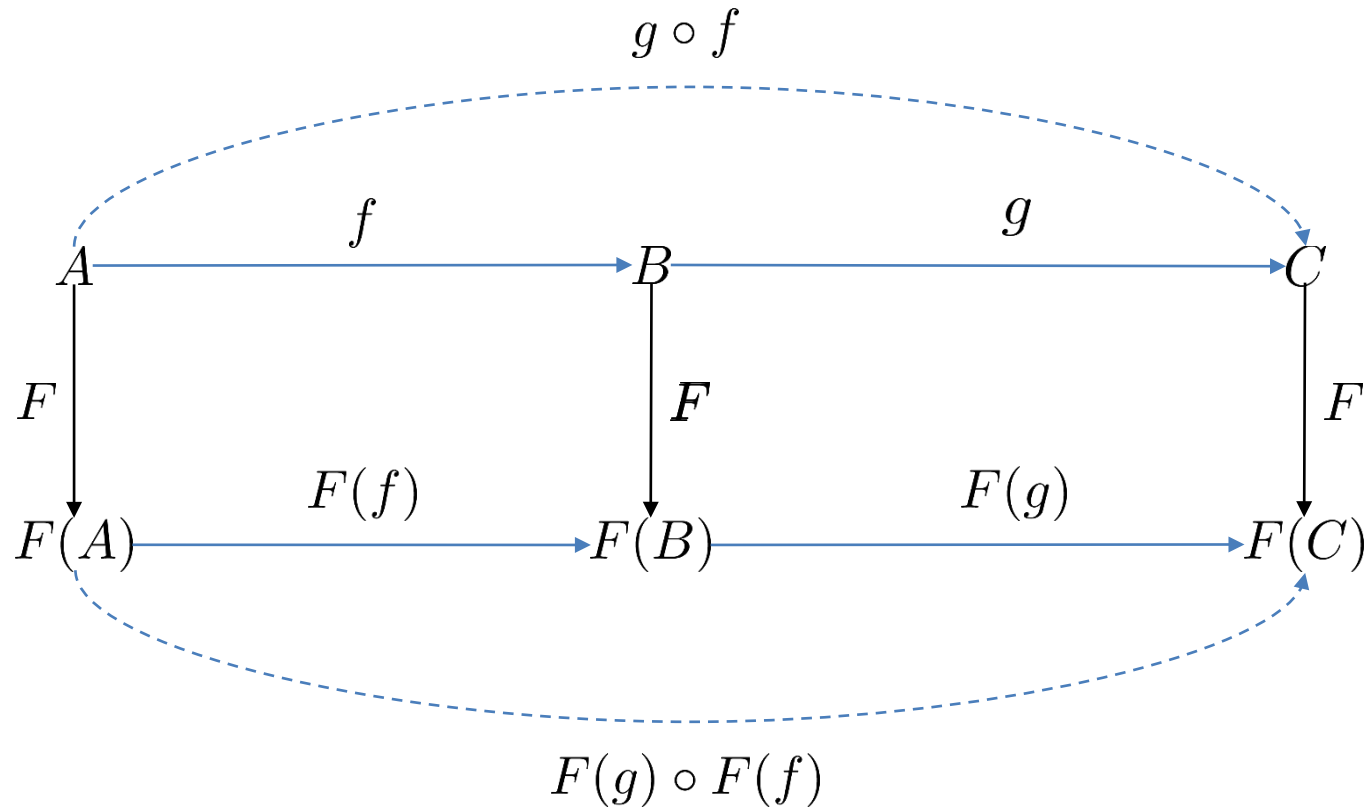
- $F$  bildet jedes Objekt  $A \in ob(\mathcal{C})$  in ein Objekt  $F(A) \in ob(\mathcal{D})$  ab
- $F$  bildet jeden Morphismus  $f : A \rightarrow B \in hom(\mathcal{C})$  in einen Morphismus  $F(f) : F(A) \rightarrow F(B) \in hom(\mathcal{D})$  ab, wobei folgende Bedingungen gelten:
  - $F(1_A) =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



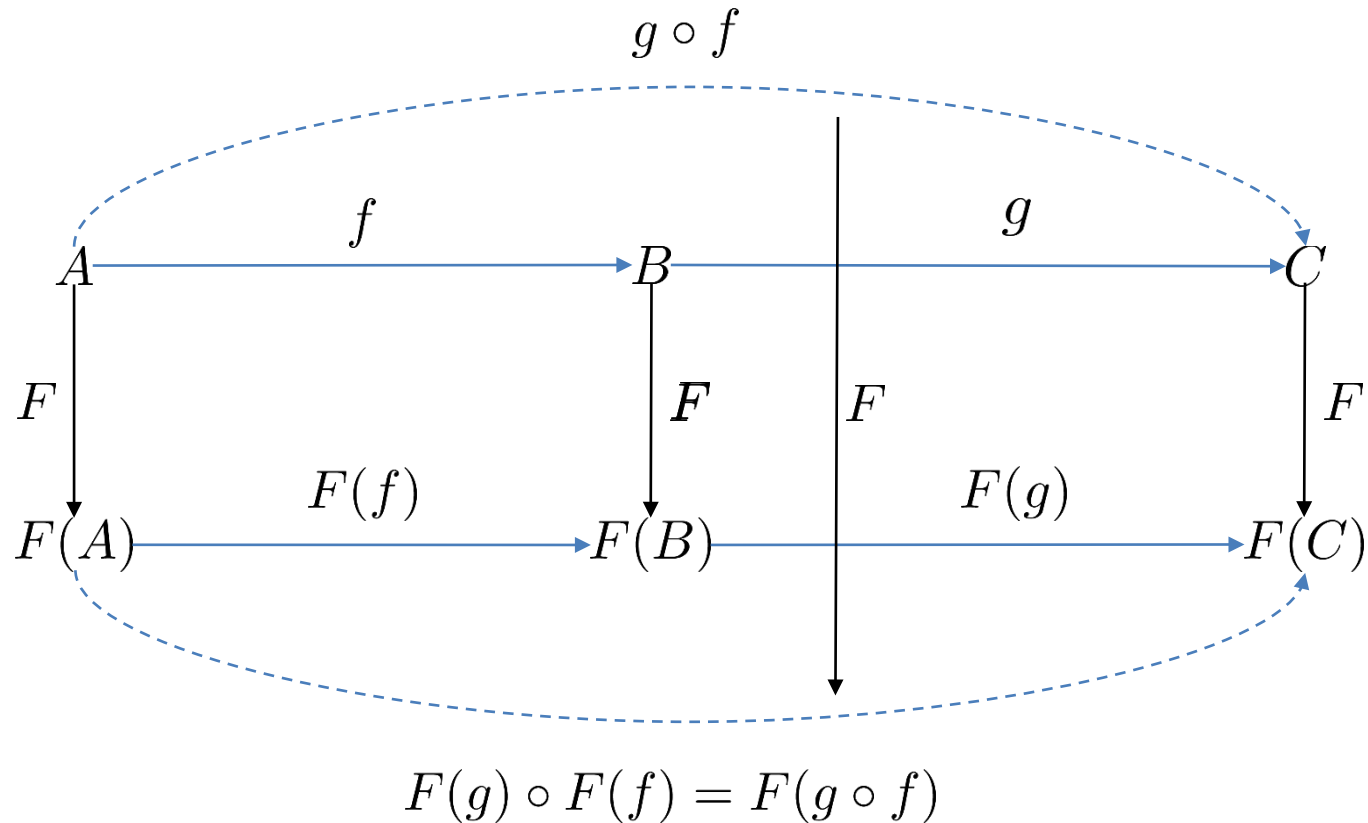
# Funktor



# Funktor



## Funktor

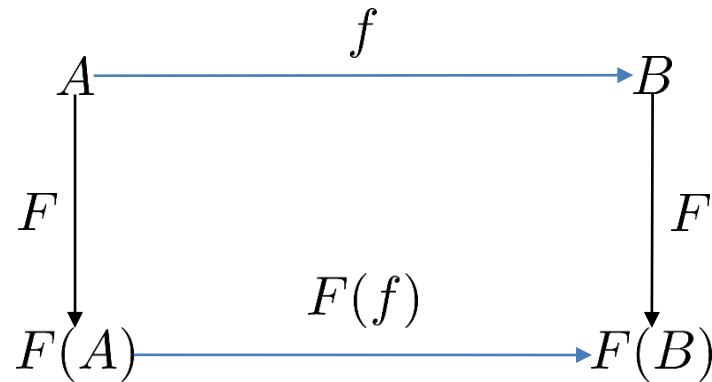


Funktoren sind Homomorphismen zwischen Kategorien

# Haskell Funktor

```

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



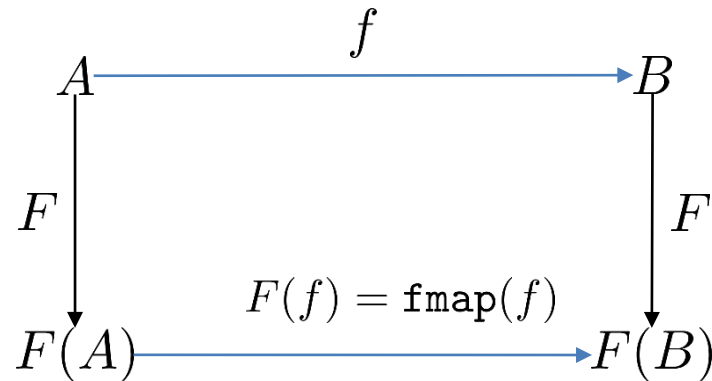


## Haskell Funktor

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

```

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

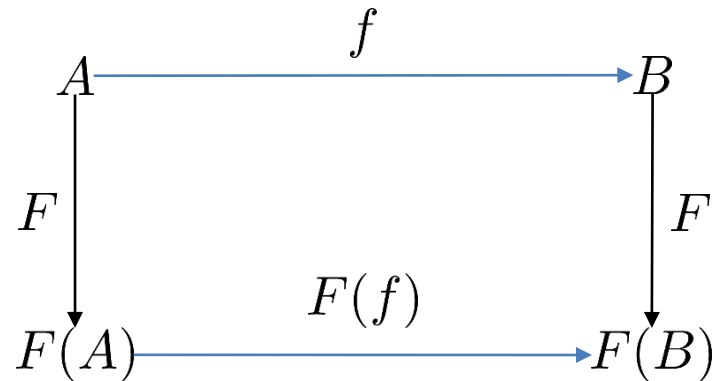


## Haskell Funktor

! In diesen Folien schreiben wir 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
```

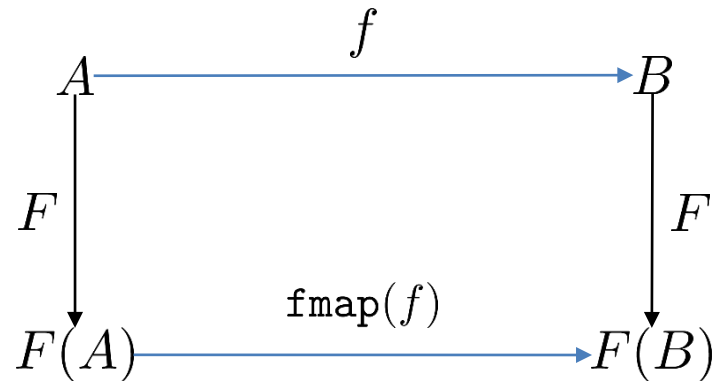


## Haskell Funktor

```

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

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



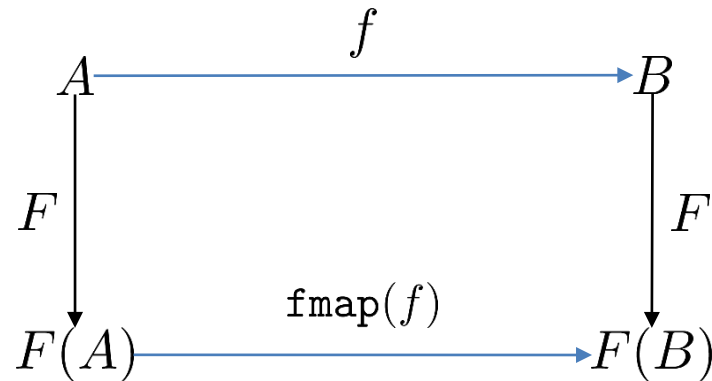
## Haskell Funktor

```

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

- $\text{fmap id} = \text{id}$
- $\text{fmap } (g \cdot f) = (\text{fmap } g) \cdot (\text{fmap } f)$

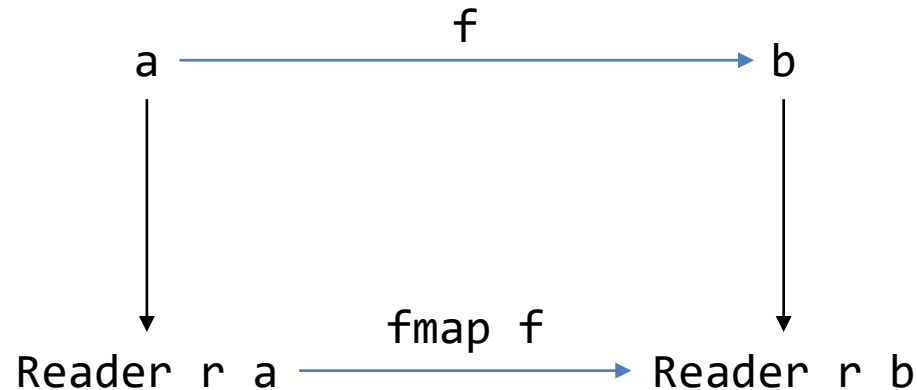
Jetzt wissen wir, woher diese  
semantischen Bedingungen  
kommen



# Reader-Funktor (Leserfunktork)

```
(->) r :: * -> *
```

```
type Reader r a = r -> a, Reader r :: * -> *
```



```
f :: a -> b, g :: Reader r a = r -> a
```

```
fmap f g = f . g
```

```
fmap :: (a -> b) -> Reader r a -> Reader r b
```

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

# Varianz

```
type Reader r a = r -> a, Reader r :: * -> *
```

Reader r ist ein Funktor mit Parameter a im Zieltyp (codomain). Was passiert, wenn wir versuchen, einen Funktor im Argumenttyp (domain) zu machen?

Wir versuchen es:

```
type Op r a = a -> r, Op r :: * -> *
```

Wir müssen nun die Funktion fmap spezifizieren, mit dem Typ:

```
fmap :: (a -> b) -> Op r a -> Op r b
```

Also

```
fmap :: (a -> b) -> (a -> r) -> (b -> r)
```

**Es gibt aber keine Funktion mit diesem Typ!**

# Kovarianz und Kontravarianz

Bei Funktionstypen spielen Argument (domain) und Ziel (codomain) unterschiedliche Rollen:

- Semantisch: domain = input, codomain = output

Mengentheoretisch:

$$A \rightarrow B = \{f \mid \forall a \in A. (f \ a) \in B\}$$
$$C \subseteq A, B \subseteq D \Rightarrow A \rightarrow B \subseteq C \rightarrow D$$

**Subtypen:**

$$\text{Real} \rightarrow \text{Int} \leq \text{Int} \rightarrow \text{Real}$$


# Kovarianz und Kontravarianz

Bei Funktionstypen spielen Argument (domain) und Ziel (codomain) unterschiedliche Rollen:

- Semantisch: domain = input, codomain = output
- Logisch: domain = Hypothese, codomain = Konklusion

Aus der Logik wissen wir: Eine Aussage mit stärkerer Konklusion ist eine stärkere Aussage („*Kovarianz*“, covariance). Eine Aussage mit stärkerer Hypothese ist eine schwächere Aussage („*Kontravarianz*“, contravariance).

Logisch:

$$C \Rightarrow A, B \Rightarrow D \quad \Rightarrow \quad (A \Rightarrow B) \Rightarrow (C \Rightarrow D)$$




# Kovarianz und Kontravarianz

- $C \leq A, B \leq D \Rightarrow A \rightarrow B \leq C \rightarrow D$
- $C \Rightarrow A, B \Rightarrow D \Rightarrow (A \Rightarrow B) \Rightarrow (C \Rightarrow D)$

Dabei haben wir eine Gleichstellung von Funktionstypen ( $\rightarrow$ ) mit logischen Implikationen ( $\Rightarrow$ ) unterstellt. Kann man das?

JA! Das heißt „Curry-Howard Isomorphismus“ und wird in der Vorlesung **LMSE I & II (Mastermodule im WS)** studiert.

Es stellt sich heraus, dass die Typentheorie eine besondere logische Theorie ist. Diese Logik ist eine Sublogik der boolschen Logik und wird als *konstruktive Logik* bezeichnet.

# Kann nützlich sein, zu wissen! Erinnerung: Varianz

Wir müssen nun die Funktion `fmap` spezifizieren, mit dem Typ:

`fmap :: (a -> b) -> Op r a -> Op r b`

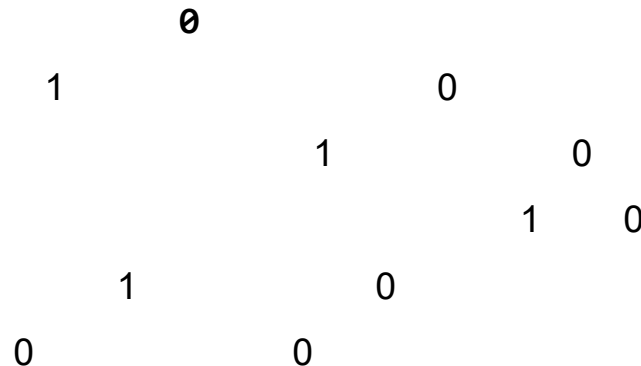
Also

`fmap :: (a -> b) -> (a -> r) -> (b -> r)`

**Es gibt aber keine Funktion mit diesem Typ!**

`(a -> b) -> ((a -> r) -> (b -> r))`

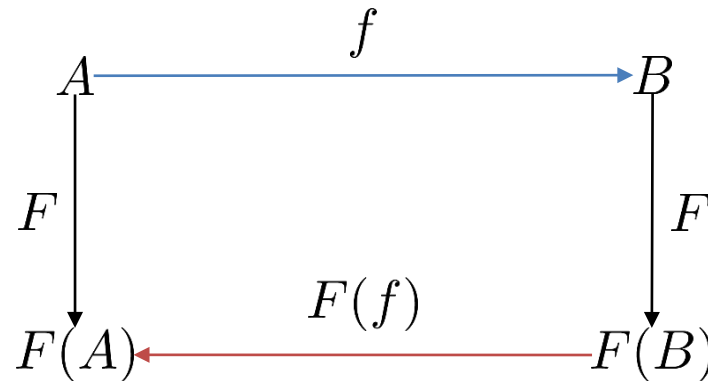
**Keine gültige Implikation!**



## Kontravarianter Funktor

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

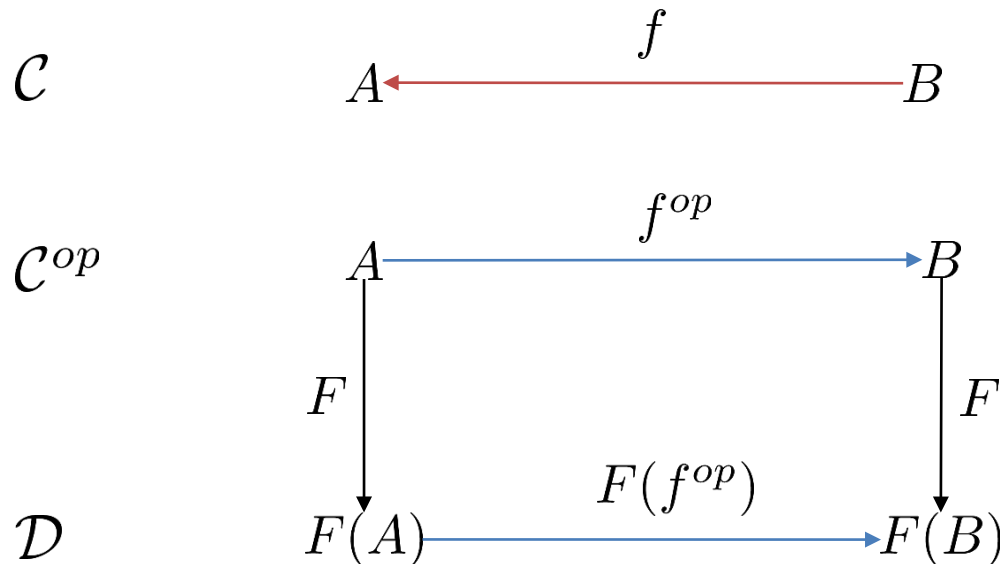
- $F$  bildet jedes Objekt  $A \in ob(\mathcal{C})$  in ein Objekt  $F(A) \in ob(\mathcal{D})$  ab
- $F$  bildet jeden Morphismus  $f : A \rightarrow B \in hom(\mathcal{C})$  in einen Morphismus  $\mathbf{F}(f) : \mathbf{F}(B) \rightarrow \mathbf{F}(A) \in hom(\mathcal{D})$  ab, wobei folgende Bedingungen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



## Kontravarianter Funktor

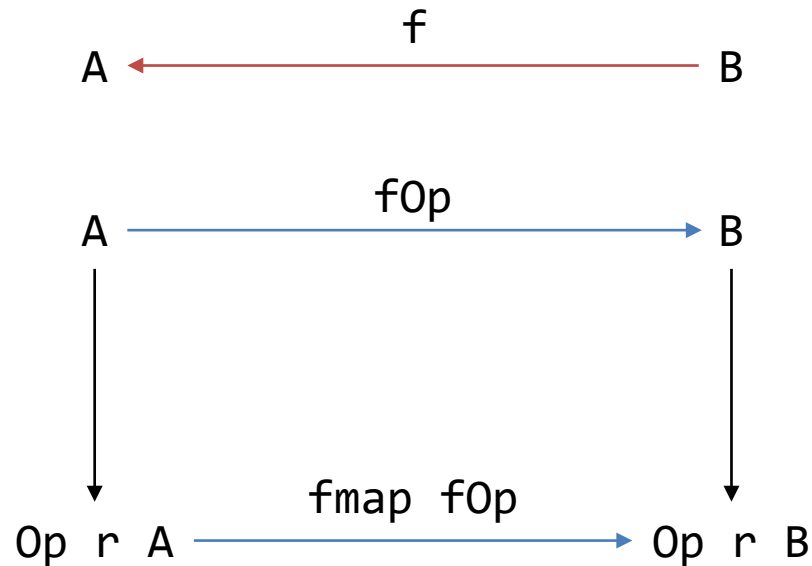
Ein *kontravarianter Funktor*  $F : \mathcal{C} \rightarrow \mathcal{D}$  kann auch als *kovarianter Funktor*  $F : \mathcal{C}^{op} \rightarrow \mathcal{D}$  in der *dualen Kategorie* aufgefasst werden.

- $\mathcal{C}^{op}$  ist die Kategorie, die aus  $\mathcal{C}$  dadurch entsteht, dass alle Morphismen in  $\mathcal{C}$  umgekehrt werden.



## Kontravarianter Funktor

```
type Op r a = a -> r, Op r :: * -> *
```



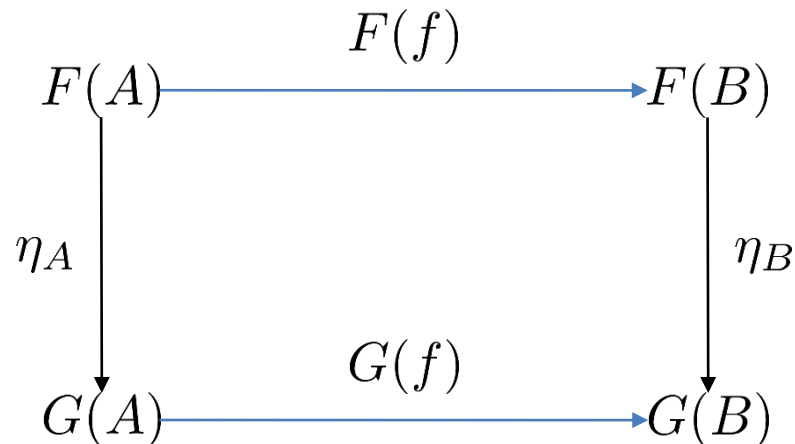
```
class Contravariant G where
```

```
  cmap :: (b -> a) -> G a -> G b
```

```
  cmap (f.g) = (cmap g).(cmap f)
```

## Natürliche Transformation

Seien  $F$  und  $G$  Funktoren von  $\mathcal{C}$  nach  $\mathcal{D}$  (Kategorien). Eine *natürliche Transformation* (eng. natural transformation)  $\eta$  von  $F$  nach  $G$  knüpft zu jedem Objekt  $A$  in  $\mathcal{C}$  einen Morphismus  $\eta_A : F(A) \rightarrow G(A)$  in  $\mathcal{D}$ , so daß es für jeden Morphismus  $f : A \rightarrow B$  in  $\mathcal{C}$  gilt:  $\eta_B \circ F(f) = G(f) \circ \eta_A$ . Mit anderen “Worten”, das Diagramm kommutiert:



Na klar doch: seien  $\mathcal{C}$  und  $\mathcal{D}$  Kategorien, dann können wir die *Funktorkategorie*  $\mathcal{D}^{\mathcal{C}}$  bilden: Objekte sind die Funktoren von  $\mathcal{C}$  nach  $\mathcal{D}$ , Morphismen sind natürliche Transformationen zwischen solchen.

# Natürliche Transformation in Haskell

$F = []$ ,  $\text{fmap } f [] = []$ ,  $\text{fmap } f (x:xs) = (f x) : \text{fmap } f xs$

$G = \text{Maybe}$ ,  $\text{fmap } f \text{ Nothing} = \text{Nothing}$ ,  $\text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta} :: [a] \rightarrow \text{Maybe } a$ ,  $\text{eta} [] = \text{Nothing}$ ,  $\text{eta } (x:xs) = \text{Just } x$

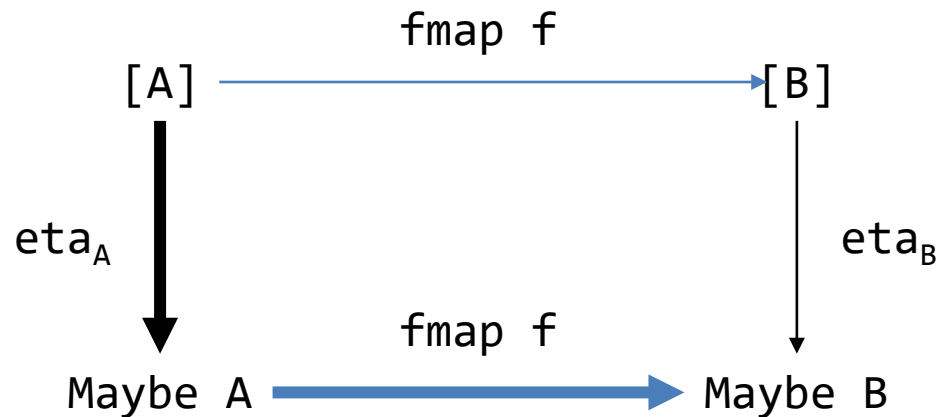


# Natürliche Transformation in Haskell

$F = []$ ,  $\text{fmap } f [] = []$ ,  $\text{fmap } f (x:xs) = (f x) : \text{fmap } f xs$

$G = \text{Maybe}$ ,  $\text{fmap } f \text{ Nothing} = \text{Nothing}$ ,  $\text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta} :: [a] \rightarrow \text{Maybe } a$ ,  $\text{eta} [] = \text{Nothing}$ ,  $\text{eta } (x:xs) = \text{Just } x$



$\text{fmap } f . \text{eta}_A$

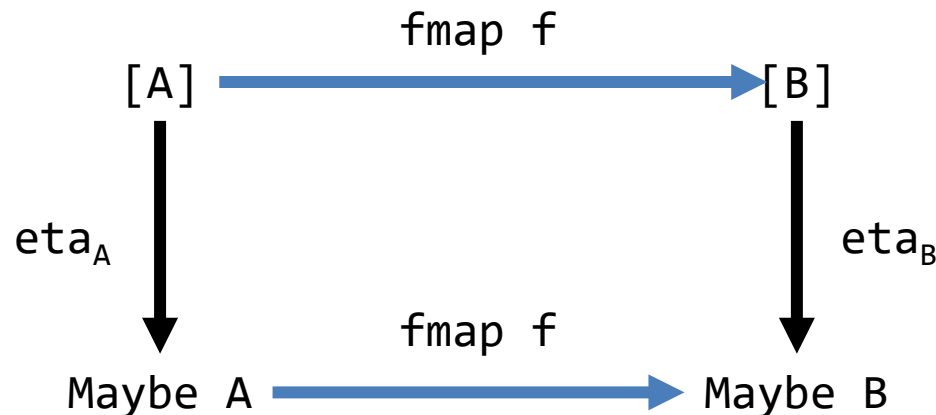


# Natürliche Transformation in Haskell

$F = []$ ,  $\text{fmap } f [] = []$ ,  $\text{fmap } f (x:xs) = (f x) : \text{fmap } f xs$

$G = \text{Maybe}$ ,  $\text{fmap } f \text{ Nothing} = \text{Nothing}$ ,  $\text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta} :: [a] \rightarrow \text{Maybe } a$ ,  $\text{eta} [] = \text{Nothing}$ ,  $\text{eta } (x:xs) = \text{Just } x$



$\text{fmap } f . \text{eta}_A$

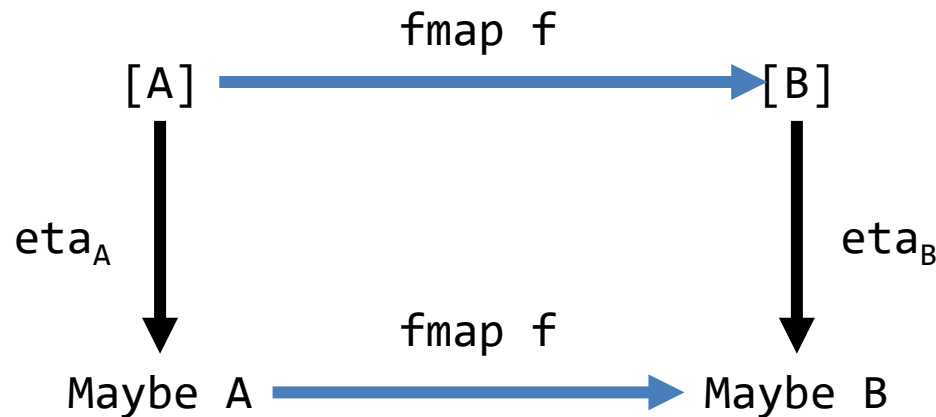
$\text{eta}_B . \text{fmap } f$

# Natürliche Transformation in Haskell

$F = []$ ,  $\text{fmap } f [] = []$ ,  $\text{fmap } f (x:xs) = (f x) : \text{fmap } f xs$

$G = \text{Maybe}$ ,  $\text{fmap } f \text{ Nothing} = \text{Nothing}$ ,  $\text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta} :: [a] \rightarrow \text{Maybe } a$ ,  $\text{eta} [] = \text{Nothing}$ ,  $\text{eta } (x:xs) = \text{Just } x$



$$\text{fmap } f . \text{eta}_A = \text{eta}_B . \text{fmap } f$$

# Natürliche Transformation in Haskell

$F = []$ ,  $\text{fmap } f [] = []$ ,  $\text{fmap } f (x:xs) = (f x) : \text{fmap } f xs$

$G = \text{Maybe}$ ,  $\text{fmap } f \text{ Nothing} = \text{Nothing}$ ,  $\text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta} :: [a] \rightarrow \text{Maybe } a$ ,  $\text{eta} [] = \text{Nothing}$ ,  $\text{eta } (x:xs) = \text{Just } x$

$\text{fmap } f . \text{eta}_A ([] ) = \text{fmap } f (\text{Nothing}) = \text{Nothing}$

$\text{fmap } f . \text{eta}_A (x:xs) = \text{fmap } f (\text{Just } x) = \text{Just } (f x)$

$\text{eta}_B . \text{fmap } f ([] ) = \text{eta}_B ([] ) = \text{Nothing}$

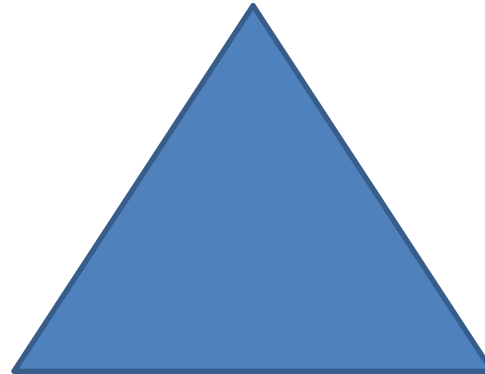
$\text{eta}_B . \text{fmap } f (x:xs) = \text{eta}_B ((f x) : \text{fmap } f xs) = \text{Just } (f x)$

# Kategorientheorie: Algebra der funktionalen Programmierung

- Theorie der Effekte (side effects, nondeterminism, control operations, CPS-transform, ...)
- Generalisierte Theorie der Faltungen (catamorphisms, anamorphisms, hylomorphisms, ...)
  - Erik Meijer et al.: *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. Functional Programming Languages and Computer Architecture, 1991. ... ETC.
- ... ..
- ... ..
- Funktionale Theorie des maschinellen Lernens
  - Fong, B., Spivak, D., & Tuyéras, R. (2019, June). *Backprop as functor: A compositional perspective on supervised learning*. In 2019 Logic in Computer Science (LICS). ... ETC.

# Trinität

Typtheorie (Lambda Kalkül)



Mathematische Logik

Kategorientheorie

# Ausblick

FuPro

...und jetzt...???

# Mastermodule LMSE I & II

## Logische Methoden des Software Engineering

- Weitergehende Typtheorie
  - Theorie höherer Typsysteme in Programmiersprachen
  - Theorie von Theorembeweisern (z.B. Coq)
  - Inhabitationsproblem und Synthese
  - Curry-Howard Isomorphismus und konstruktive Logik
  - Grundlagentheorie der mathematischen Logik
- Weitergehende Theorie des Lambda-Kalküls
  - Beweise von Fundamentalsätzen (z.B. Church-Rosser)
  - Ausdrucksmächtigkeit
  - Normalisierung und Typen

# Und mehr...?

- Bachelor:
  - Proseminar: Perlen der Funktionalen Programmierung (SoSe, Dr. Dudenhefner)
    - [Proseminar Perlen der funktionalen Programmierung - Lehrstuhl 14 für Software Engineering - Fakultät für Informatik - TU Dortmund \(tu-dortmund.de\)](https://tu-dortmund.de)
  - Proseminar: Lösungsansätze für das Expression Problem (SoSe, Prof. Rehof)
  - MNP: Modellierung Nebenläufiger Prozesse (SoSe, Prof. Rehof)
  - Abschlussarbeiten (SoSe/WiSe, LS14 / AG SEAL)
- Master:
  - Seminar: Principles of Programming Languages (WiSe, Prof. Rehof)
  - Vertiefung: Logische Methoden des Software Engineering 1 & 2 (WiSe, Prof. Rehof)
  - Vertiefung: Aktuelle Themen im logikbasierten Software Engineering (SoSe, Dr. Dudenhefner)
  - Vertiefung: Type Systems for Correctness and Security (WiSe, Prof. Hermann)
  - Projektgruppen (SoSe/WiSe, Laarmann & Dr. Hildebrand)
  - Abschlussarbeiten (SoSe/WiSe, LS14 / AG SEAL)



haskell

2 Jobs für die Suche nach haskell

Jobmailer

Merkliste

Filtern und Sortieren

## Machine Learning Spezialist (m/w/d)

05.07.2023

★ Merken

## Software Engineer (Development Lead) - Digital Products

30.06.2023

★ Merken

← Startseite

1

weitere Jobs >

Passende Jobs zu Ihrer Suche ...

... immer aktuell und kostenlos per E-Mail.

Ihre E-Mail-Adresse

Jobs kostenlos per E-Mail

Sie können den Suchauftrag jederzeit abbestellen.

Es gilt unsere Datenschutzerklärung. Sie erhalten passende Angebote per E-Mail. Sie können sich jederzeit wieder kostenlos abmelden.

In einem Startup-ähnlichen Umfeld, in dem die agilen Werte und enge Zusammenarbeit der Schlüssel zum Erfolg sind, gestalten wir die Zukunft.

### Aufgaben

- > Als Software Engineer (Development Lead) - Digital Products (m/w/d) bist du motiviert, deine Expertise und Erfahrung in die Weiterentwicklung junger Talente einzubringen
- > Im Team konzipierst und entwickelst du performante, skalierbare und sichere Anwendungen mit den Technologien und Programmiersprachen der Wahl
- > Du arbeitest in einem cross-funktionalen XP-Team und fühlst dich im Pair Programming sehr wohl
- > Du stellst durch Testautomatisierung und TDD sicher, dass unsere Produkte den höchsten Qualitätsstandards genügen
- > Durch Monitoring und Analyse des Laufzeitverhaltens deiner Software findest du Probleme frühzeitig

### Qualifikation

- > Du hast ein Studium in (Wirtschafts-)Informatik, Wirtschaftsingenieurwesen erfolgreich abgeschlossen bzw. kannst eine vergleichbare Ausbildung mit einschlägiger Berufserfahrung vorweisen
- > Du bringst sehr gute Kenntnisse in einer objektorientierten Programmiersprache wie z.B. Java, C-Sharp oder C++ und in mindestens einer der Programmiersprachen Go, Rust, Haskell, Kotlin oder Erlang mit
- > Du verfügst über mehrjährige Praxiserfahrung im Design und in der Implementierung von hoch skalierbaren, verteilten und cloud-basierten Systemen sowie in TDD und CI/CD
- > Du hast idealerweise Vorerfahrung im Incident Management in einem DevOps-Team
- > Eine ausgeprägte Kommunikationsfähigkeit in Englisch rundet dein Profil ab

### Deine Vorteile

bietet dir ein umfassendes Angebot an Leistungen wie flexible und hybride Arbeitszeitmodelle, Gesundheitsmaßnahmen und Unterstützung bei Familie und Pflege, ein attraktives Grundgehalt und Erfolgsbeteiligungen, verschiedene Mobilitätsangebote sowie vieles mehr unter

Quelle: <https://jobs.heise.de/> (Abrufdatum: 14.07.23)

haskell

2 Jobs für die Suche nach haskell

Jobmailer

Merkliste

Filtern und Sortieren

**Informatiker/in, Ingenieur/in Informationstechnik, Physiker/in o. ä. (w/m/d) Forschung und Entwicklung im Bereich maritimer Lagebildsysteme**

11.07.2024

★ Merken

**Informatiker/in, Ingenieur/in Informationstechnik, Physiker/in o. ä. (w/m/d) Forschung und Entwicklung im Bereich maritimer Lagebildsysteme**

30.06.2024

★ Merken

< Startseite

1

weitere Jobs >

Passende Jobs zu Ihrer Suche ...

... immer aktuell und kostenlos per E-Mail.

Ihre E-Mail-Adresse

Jobs kostenlos per E-Mail

Sie können den Suchauftrag jederzeit abbestellen.

Es gilt unsere Datenschutzerklärung. Sie erhalten passende Angebote per E-Mail. Sie können sich jederzeit wieder kostenlos abmelden.

Realisierung dieser Daten mit modernen Design-Konzepten generiert zu werden. Aufgaben, eine gemeinsame Sprache für die Kommunikation zwischen den Sensoren zu etablieren. Diese spannende Pionierarbeit für die Zukunft der maritimen Sicherheit wartet darauf, von dir bearbeitet zu werden.

Deine Arbeit wird außerdem ein wichtiger Teil eines großen und wichtigen Projekts sein, bei dem diese neuen Sensorsysteme in unser maritimes Lagebild integriert werden – ein weiterer Fortschritt für die maritime Sicherheit.

Vielleicht denkst du auch darüber nach zu promovieren? **[REDACTED]** unterstützen wir dich auch auf deinem akademischen Weg!

Werde Teil unseres jungen, kollegialen und dynamischen Teams und übernehme folgende Aufgaben:

- Erstellung von Machbarkeitsstudien und Konzepten für diverse Systemkonzepte
- Spezifikation von Systemarchitektur und Systemkomponenten
- Softwareentwicklung für den Aufbau verschiedener Systemkomponenten und deren Integration in ein Gesamtlagebild
- Erprobung des Systems gemeinsam mit Instrument- und Datenwissenschaftlerinnen und -wissenschaftlern
- Publikation und Präsentation von Forschungsergebnissen

## Das erwarten wir von Ihnen:

- abgeschlossenes wissenschaftliches Hochschulstudium (Universitätsdiplom oder Master) der Informatik (z. B. Angewandte Informatik, Medieninformatik), in einem ingenieurwissenschaftlichen Bereich (z. B. Elektrotechnik), einem naturwissenschaftlichen Bereich (z. B. Physik) oder andere für die Tätigkeit relevante Studiengänge
- ausgeprägte Kenntnisse gängiger Programmierparadigmen und -sprachen (insbesondere objektorientierte und funktionale Sprachen, wie z. B. C++, Java, Python, Scala, Haskell oder Rust)
- einschlägige Erfahrungen auf dem Gebiet der Sensornetze oder anderer vergleichbarer Systeme
- ausgeprägte Kommunikations- und Kooperationsbereitschaft sowie Teamfähigkeit
- Fähigkeit und Bereitschaft sich fehlende Kenntnisse bei Bedarf kurzfristig anzueignen
- Kreativität, Eigeninitiative und Zuverlässigkeit

Quelle: <https://jobs.heise.de/> (Abrufdatum: 12.07.24)

Job, Firma oder Job-ID  
**Haskell**



Ort oder PLZ

**Jobs finden**

## Oops ...

Wir konnten leider keine passenden Jobs zu deiner Suche nach **Haskell** finden.



















Versuche stattdessen das Folgende:

- Überprüfe deinen Suchbegriff auf mögliche Tippfehler.
- Nutze eher allgemeine und weniger spezifische Suchbegriffe.
- Vermeide Abkürzungen und verwende stattdessen die ausgeschriebene Version.
- Starte eine neue Suche mit einem alternativen Suchbegriff.
- Überprüfe deine gewählten Filter.

**zurück**

Quelle: <https://jobs.heise.de/> (Abrufdatum: 16.07.25)

Gründe?

Jul 2025	Jul 2024	Change	Programming Language		Ratings	Change
1	1			Python	26.98%	+10.85%
2	2			C++	9.80%	-0.53%
3	3			C	9.65%	+0.16%
4	4			Java	8.76%	+0.17%
5	5			C#	4.87%	-1.85%
6	6			JavaScript	3.36%	-0.43%
7	7			Go	2.04%	-0.14%
8	8			Visual Basic	1.94%	-0.13%
9	24	⬆		Ada	1.77%	+0.99%
10	11	⬆		Delphi/Object Pascal	1.77%	-0.12%
11	30	⬆		Perl	1.76%	+1.10%
12	9	⬇		Fortran	1.67%	-0.38%
13	10	⬇		SQL	1.39%	-0.65%
14	16	⬆		PHP	1.28%	+0.14%
15	22	⬆		R	1.25%	+0.42%
16	12	⬇		MATLAB	1.11%	-0.23%
17	15	⬇		Scratch	1.06%	-0.09%
18	13	⬇		Rust	1.01%	-0.17%
19	18	⬇		Assembly language	0.94%	-0.18%
20	20			Kotlin	0.90%	-0.15%

Quelle:

<https://www.tiobe.com/tiobe-index/>

(Abrufdatum: 16.07.25)




















Position	Programming Language	Ratings
21	Swift	0.85%
22	COBOL	0.83%
23	Ruby	0.76%
24	Lisp	0.75%
25	Prolog	0.73%
26	Classic Visual Basic	0.63%
27	SAS	0.62%
28	Dart	0.61%
29	Lua	0.46%
30	(Visual) FoxPro	0.44%
31	Haskell	0.43%
32	Objective-C	0.42%
33	GAMS	0.42%
34	Scala	0.41%
35	Julia	0.41%
36	VBScript	0.37%
37	TypeScript	0.28%
38	ABAP	0.27%
39	PL/SQL	0.24%
40	D	0.19%
41	Solidity	0.18%
42	V	0.18%
43	Bash	0.18%
44	Elixir	0.17%
45	PowerShell	0.16%
46	Awk	0.16%
47	ML	0.15%
48	X++	0.14%
49	RPG	0.14%
50	LabVIEW	0.13%

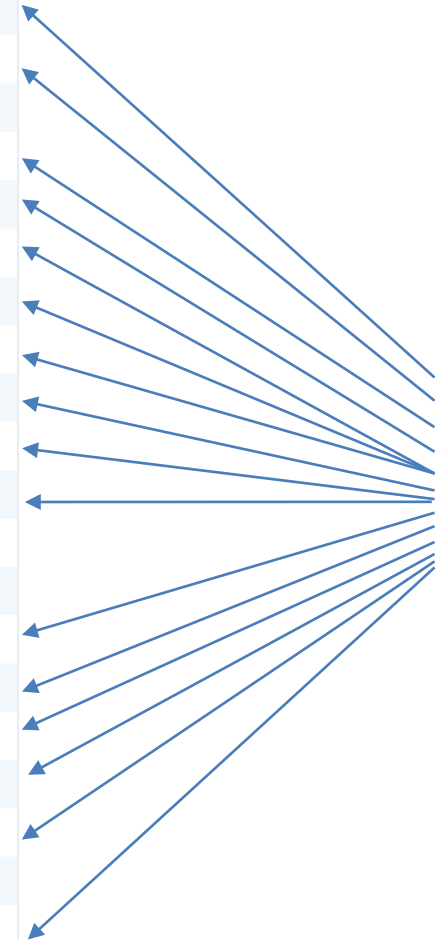
<https://www.tiobe.com/tiobe-index/>

(Abrufdatum: 16.07.25)

# Ideen & Techniken aus der Funktionalen Programmierung

- $\lambda$ -Ausdrücke
- Listenkomprehensionen
- Faltungen
- Funktoren & Monaden
- Church Kodierungen
- ...

Jul 2025	Jul 2024	Change	Programming Language		Ratings	Change
1	1			Python	26.98%	+10.85%
2	2			C++	9.80%	-0.53%
3	3			C	9.65%	+0.16%
4	4			Java	8.76%	+0.17%
5	5			C#	4.87%	-1.85%
6	6			JavaScript	3.36%	-0.43%
7	7			Go	2.04%	-0.14%
8	8			Visual Basic	1.94%	-0.13%
9	24	⬆		Ada	1.77%	+0.99%
10	11	⬆		Delphi/Object Pascal	1.77%	-0.12%
11	30	⬆		Perl	1.76%	+1.10%
12	9	⬇		Fortran	1.67%	-0.38%
13	10	⬇		SQL	1.39%	-0.65%
14	16	⬆		PHP	1.28%	+0.14%
15	22	⬆		R	1.25%	+0.42%
16	12	⬇		MATLAB	1.11%	-0.23%
17	15	⬇		Scratch	1.06%	-0.09%
18	13	⬇		Rust	1.01%	-0.17%
19	18	⬇		Assembly language	0.94%	-0.18%
20	20			Kotlin	0.90%	-0.15%



$\lambda$

Quelle:

<https://www.tiobe.com/tiobe-index/>

(Abrufdatum: 16.07.25)





### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```

>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
    
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

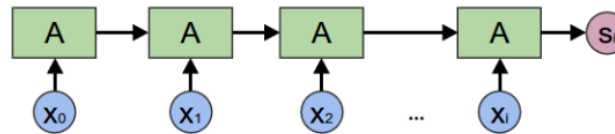
```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

Quellen: <https://docs.python.org/3/tutorial/datastructures.html> &  
<https://www.python.org/static/img/python-logo@2x.png>

(Abrufdatum: 14.07.22)

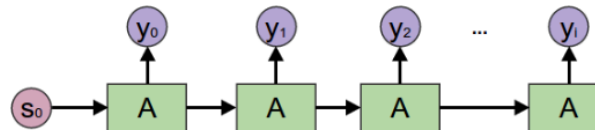
- **Encoding Recurrent Neural Networks** are just folds. They're often used to allow a neural network to take a variable length list as input, for example taking a sentence as input.



fold = Encoding RNN 1

Haskell: `foldl a s`

- **Generating Recurrent Neural Networks** are just unfolds. They're often used to allow a neural network to produce a list of outputs, such as words in a sentence.



unfold = Generating RNN

Haskell: `unfoldr a s`

Quelle:

<https://colah.github.io/posts/2015-09-NN-Types-FP/>

(Abrufdatum: 14.07.22)



So monad support is built-in in the C# language. As discussed in the LINQ query expression pattern part, SelectMany enables multiple from clauses, which can chain operations together to build a workflow, for example:

```

internal static void Workflow<T1, T2, T3, T4>(
    Func<IEnumerable<T1>> source1,
    Func<IEnumerable<T2>> source2,
    Func<IEnumerable<T3>> source3,
    Func<T1, T2, T3, IEnumerable<T4>> source4)
{
    IEnumerable<T4> query = from value1 in source1()
                           from value2 in source2()
                           from value3 in source3()
                           from value4 in source4(value1, value2, value3)
                           select value4; // Define query.
    query.WriteLine(); // Execute query.
}
    
```

Quellen: <https://weblogs.asp.net/dixin/category-theory-via-csharp-7-monad-and-linq-to-monads> & <https://de.wikipedia.org/wiki/C-Sharp>

(Abrufdatum: 14.07.22)

# Extensibility for the Masses

## Practical Extensibility with Object Algebras

Bruno C.d.S. Oliveira<sup>1</sup> and William R. Cook<sup>2</sup>

<sup>1</sup> National University of Singapore  
bruno@ropas.snu.ac.kr

<sup>2</sup> University of Texas, Austin  
wcook@cs.utexas.edu

**Abstract.** This paper presents a new solution to the expression problem (EP) that works in OO languages with simple generics (including Java or C#). A key novelty of this solution is that advanced typing features, including F-bounded quantification, wildcards and variance annotations, are not needed. The solution is based on object algebras, which are an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the VISITOR pattern, but without many of its drawbacks: they are

[Oliveira, Bruno C. D. S., and William R. Cook. "Extensibility for the masses: Practical extensibility with object algebras." European Conference on Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.](#)