Tutorium 04: λ-Kalkül und Haskell

Paul Brinkmeier

22. November 2022

Tutorium Programmierparadigmen am KIT

Übungsblatt 3

Wiederholung: `zipWith`

```
xs = repeat 1
  => [1, 1, 1, 1, 1, 1, 1, ...]
ys = [1, 2, 3, 4, 5, 6, 7]

zs = zipWith (/) xs ys
  => [1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/6]
```

- `zipWith op xs ys` wendet `op` paarweise auf die Elemente von `xs` und `ys` an.
- Typisches Beispiel: `zipWith (*)` (Hadamard-Produkt)

Wiederholung: `zipWith`

```
xs = repeat 1
  => [1, 1, 1, 1, 1, 1, 1, ...]
ys = [1, 2, 3, 4, 5, 6, 7]

zs = zipWith (/) xs ys
  => [1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/6]
```

- `zipWith op xs ys` wendet `op` paarweise auf die Elemente von `xs` und `ys` an.
- Typisches Beispiel: `zipWith (*)` (Hadamard-Produkt)
- Wie die meisten Listenkombinatoren funktioniert `zipWith` auch mit unendlichen Listen:
  - Beide Eingaben unendlich $\implies$ Ausgabe unendlich
  - Anderenfalls hat die Ausgabe die Länge der kürzeren Eingabe

1 — Streams

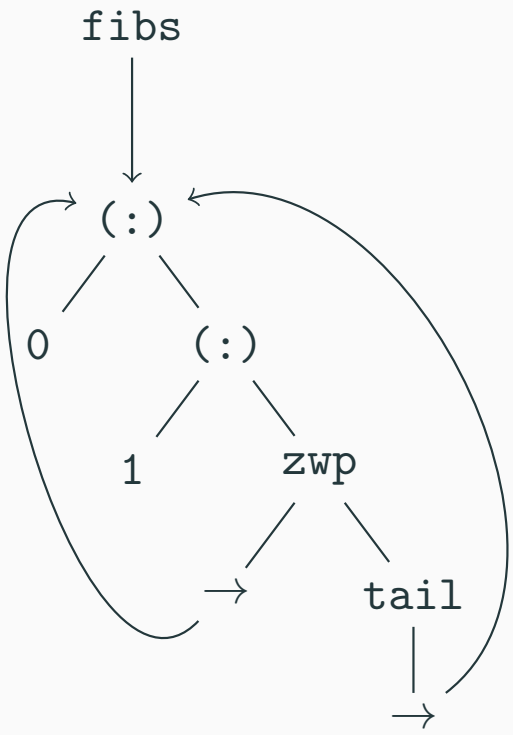
```
module Fibs where


fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Auswertung:

```
-- zwp = zipWith (+)
  0 : 1 : zwp fibs         (tail fibs)
= 0 : 1 : zwp (0 : 1 : _) (1 : _)
= 0 : 1 : 0 + 1 : zwp (1 : 0 + 1 : _) (0 + 1 : _)
= 0 : 1 : 1 : zwp (1 : 1 : _) (1 : _)
= 0 : 1 : 1 : 2 : zwp(1 : 2 : _) (2 : _) = ...
```

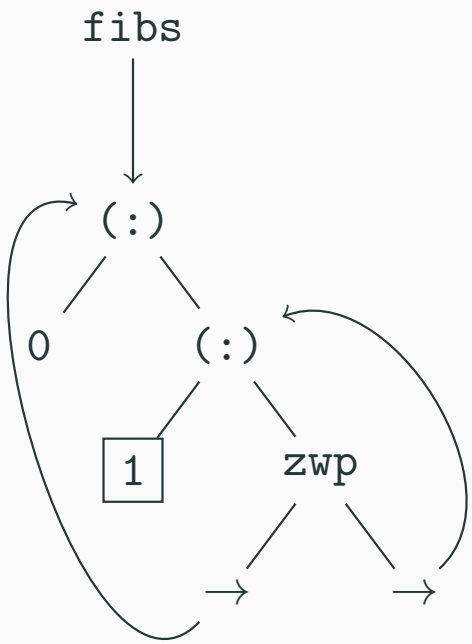
1 — Streams

```
fibs = 0 : (1 : zipWith (+) fibs (tail fibs))
```



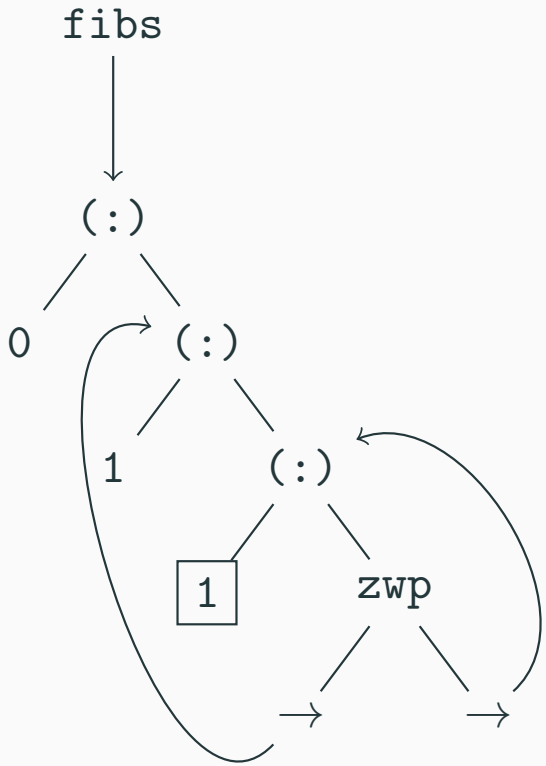
- `zwp = zipWith (+)`
- Laufzeit: Alle Vorkommen von `fibs` beziehen sich auf dasselbe Speicherobjekt (Sharing)
- `tail (x:xs) = xs`

1 — Streams



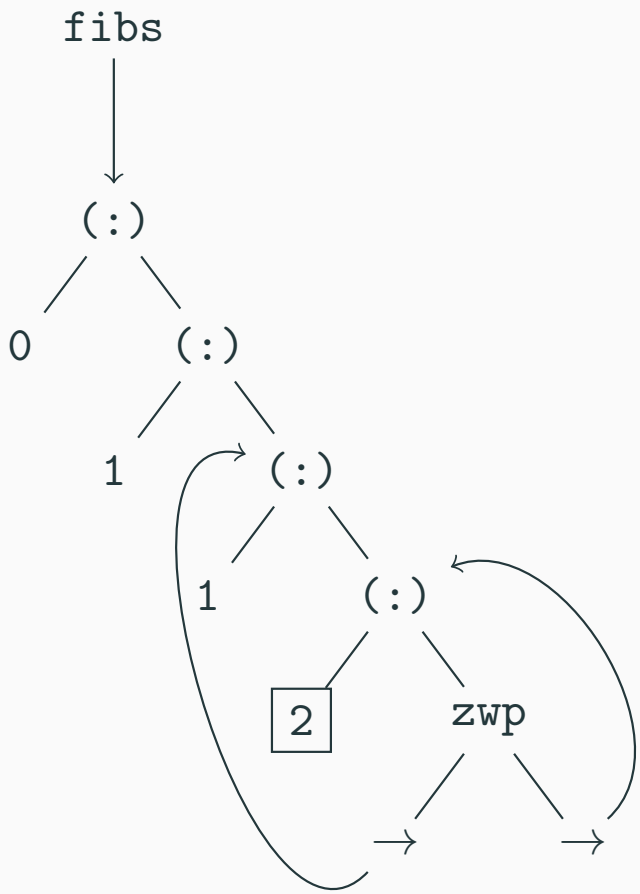
- `tail (x:xs) = xs`
- `fibs !! 1 == 1`
- Keine Berechnung notwendig
- `fibs !! 3`?

1 — Streams



- `zwp (x:xs) (y:ys) = (x + y) : zwp xs ys`
- `fibs !! 2 == 1`
- `fibs !! 3`?

1 — Streams



- `zwp (x:xs) (y:ys) = (x + y) : zwp xs ys`
- `fibs !! 3 == 2`
- `zwp` verschiebt nur Zeiger auf bestehendes Objekt, `fibs` wird nur einmal berechnet und steht ab da zur Verfügung

:sprint

```
*Fibs> :sprint fibs
fibs = _
*Fibs> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
*Fibs> :sprint fibs
fibs = 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : _
```

- `:sprint a` gibt aktuelle Speicherrepräsentation für `a` aus
- `_` steht dabei für „noch nicht ausgewertet“
- $\leadsto$ praktisch für Debugging unendlicher Listen

3 — Stream-Kombinatoren

Gesucht für $n \in \mathbb{N}$:

$$\{p^i | p \text{ Primzahl}, 1 \leq i \leq n\}$$

Gegeben:

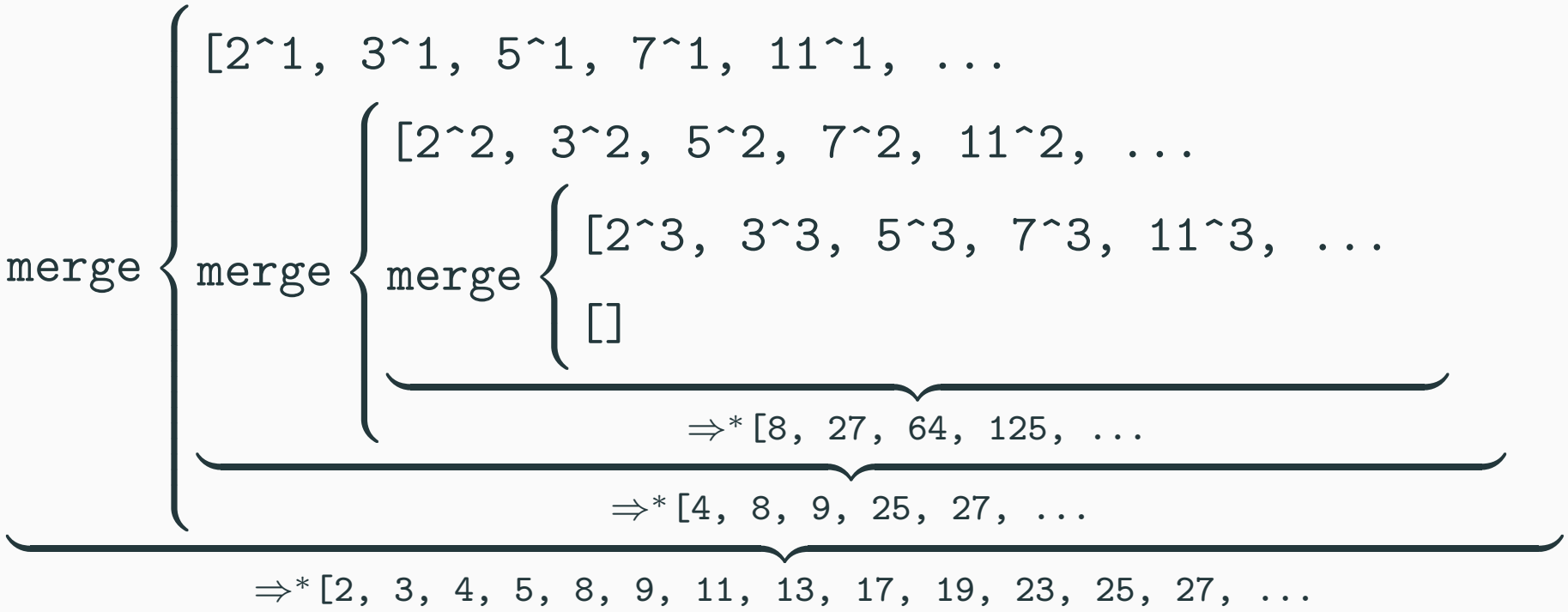
```
-- [2, 3, 5, 7, ...
primes :: [Integer]

--      Eingabelisten aufsteigend sortiert
-- ==> Ausgabeliste aufsteigend sortiert
merge :: Ord a => [a] -> [a] -> [a]
```

Schematisch sehen (für $n = 3$) unsere Eingabedaten so aus:

i	`map (^i) primes`									
1	2	3	5	7	11	13	17	19	23	...
2	4	9	25	49	121	169	289	361	529	...
3	8	27	125	343	1331	2197	4913	6859	12167	...

- `foldr` funktioniert auch mit unendlichen Listen
  $\rightsquigarrow$ Aber weiß nicht, dass `primes`/`merge xs ys` monoton.
- `foldl` funktioniert nur mit endlichen Listen
  $\Longrightarrow$ `foldl merge` über `map (^i) primes`

3 — Stream-Kombinatoren



Wiederholung: Algebraische Datentypen

Cheatsheet: Algebraische Datentypen in Haskell

- *`data`-Definitionen*, *Datenkonstruktoren*
- Algebraische Datentypen: *Produkttypen* und *Summentypen*
  - Produkttypen $\approx$ `structs` in C
  - Summentypen $\approx$ `enums`
- *Typkonstruktoren*, bspw. `[] :: * -> *`
- *Polymorphe* Datentypen, bspw. `[a]`, `Maybe a`
- Beispiel:

```
module Shape where


data Shape
  = Circle Double -- radius
  | Rectangle Double Double -- sides
  | Point -- technically equivalent to Circle 0
```

Cheatsheet: Typklassen 1

- *Klasse*, *Operationen/Methoden*, *Instanzen*
- Beispiele:
  - `Eq t`, {`(==)`, `(/=)`}, {`Eq Bool`, `Eq Int`, `Eq Char`, ...}
  - `Show t`, {`show`}, {`Show Bool`, `Show Int`, `Show Char`, ...}
- Weitere Typklassen: `Ord`, `Num`, `Enum`
- Deklaration/Implementierung:

```
module Truthy where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0
```

Cheatsheet: Typklassen 2

- *Vererbung*: Typklassen mit Voraussetzungen

```
module Truthy2 where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0

instance Truthy t => Truthy (Maybe t) where
  toBool Nothing  = False
  toBool (Just x) = toBool x
```

Spielkarten

```
module PlayingCard where

data PlayingCard = PlayingCard Suit Rank

data Suit = Hearts | Diamonds | Clubs | Spades
data Rank
  = Rank7 | Rank8 | Rank9 | Rank10
  | Jack  | Queen | King  | Ace
```

Monopolykarten

```
module Monopoly where

data MonopolyCard
  = Street String Rent Int Color
  | Station String
  | Utility String

data Rent = Rent Int Int Int Int Int

data Color
  = Brown | LightBlue | Pink   | Orange
  | Red   | Yellow    | Green | Blue
```

Boolesche Logik

```
module BoolExpr where

data BoolExpr
  = Const Bool
  | Var String
  | Neg BoolExpr
  | BinaryOp BoolExpr BinaryOp BoolExpr

data BinaryOp = AND | OR | XOR | NOR
```

Beispiele:

- $a \wedge b$ entspricht `BinaryOp (Var "a") AND (Var "b")`
- $a \vee (b \wedge 0)$ entspricht
  `BinaryOp (Var "a") AND (BinaryOp (Var "b") OR (Const False))`

λ-Kalkül

λ-Kalkül

- „Funktionales Gegenstück zur Turingmaschine“
- Gönnt Punkte in der Klausur
  - 13P. im 19SS
  - 10P. (+15P.) im 18WS
  - 20P. (+15P.) im 18SS

λ-Terme

Ein Term im λ-Kalkül hat eine der drei folgenden Formen:

Notation	**Besteht aus**	**Bezeichnung**
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ-Term	Abstraktion
$f\ a$	f, a : λ-Terme	Funktionsanwendung

- „λ-Term“: rekursive Datenstruktur
- Semantik definieren wir später

λ-Terme

Ein Term im λ-Kalkül hat eine der drei folgenden Formen:

Notation	**Besteht aus**	**Bezeichnung**
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ-Term	Abstraktion
$f\ a$	f, a : λ-Terme	Funktionsanwendung

- „λ-Term“: rekursive Datenstruktur
- Semantik definieren wir später
- Jetzt: Ergänzt das Modul `Lambda` um die fehlenden Typen
  - +Fragen zur ÜB-Korrektur

λ-Terme in Haskell

```
module Lambda where


data LambdaTerm
  = Var String  -- Variable
  | App () ()   -- Funktionsanwendung: f a
  | Abs () ()   -- Abstraktion: \p.b
```

- Variable x hat einen Variablennamen x
- Funktionsanwendung $f\ a$ hat zwei λ-Terme: Funktion f und Argument a
- Abstraktion $\lambda p.\, b$ hat Variablennamen p als Parameter und λ-Term b als Körper (Body)

Begriffe im λ-Kalkül

Begriff	**Formel**	**Bedeutung**
α-Äquivalenz	$t_1 \overset{\alpha}{=} t_2$	t_1, t_2 sind gleicher Struktur
η-Äquivalenz	$\lambda x.f\ x \overset{\eta}{=} f$	„Unterversorgung“
Freie Variablen	$fv(\lambda p.b) = b$	Menge der nicht durch λs gebundenen Variablen
Substitution	$(\lambda p.b)\,[b \to c] = \lambda p.c$	Ersetzung freier Variablen
Redex	$(\lambda p.b)\ t$	„Reducible expression“
β-Reduktion	$(\lambda p.b)\ t \Rightarrow b\,[p \to t]$	„Funktionsanwendung“

Freie Variablen

- $fv(t)$ bezeichnet die frei vorkommenden Variablen im Term t
- Frei vorkommend $\approx$ nicht durch ein λ gebunden
  - $fv(x) = \{x\}$, wenn x Variable
  - $fv(f\ x) = fv(f) \cup fv(x)$
  - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
  - $fv(\lambda x.x) = \emptyset$
  - $fv(\lambda x.y) = \{y\}$

Freie Variablen

- $fv(t)$ bezeichnet die frei vorkommenden Variablen im Term t
- Frei vorkommend $\approx$ nicht durch ein λ gebunden
  - $fv(x) = \{x\}$, wenn x Variable
  - $fv(f\ x) = fv(f) \cup fv(x)$
  - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
  - $fv(\lambda x.x) = \emptyset$
  - $fv(\lambda x.y) = \{y\}$
- Implementiert `fv :: LambdaTerm -> Set String`
  - Benutzt `Set`, `union`, `delete` und `fromList` aus `Data.Set`
  - Bspw. `fv (Abs "p"(Var "b")) == fromList ["b"]`

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t\,[a \to b]$ — Ersetze a durch b in t
- Beispiele:
  - $a\,[a \to b] = b$
  - $a\,[b \to c] = a$
  - $(f\ x)\,[f \to g]\,[x \to y] = g\ y$

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t\,[a \to b]$ — Ersetze a durch b in t
- Beispiele:
  - $a\,[a \to b] = b$
  - $a\,[b \to c] = a$
  - $(f\ x)\,[f \to g]\,[x \to y] = g\ y$
  - $(\lambda x.f\ x)\,[x \to y] = \lambda x.f\ x$ (x ist nicht frei)
  - $(\lambda x.f\ x)\,[f \to g] = \lambda x.g\ x$ (f ist frei)

Substitution

- Substitution ersetzt alle *freien* Variablen in einem Term
- $t\,[a \to b]$ — Ersetze a durch b in t
- Beispiele:
  - $a\,[a \to b] = b$
  - $a\,[b \to c] = a$
  - $(f\ x)\,[f \to g]\,[x \to y] = g\ y$
  - $(\lambda x.f\ x)\,[x \to y] = \lambda x.f\ x$ (x ist nicht frei)
  - $(\lambda x.f\ x)\,[f \to g] = \lambda x.g\ x$ (f ist frei)
- Implementiert
  `substitute :: (String, Term) -> Term -> Term`
  - `type Term = LambdaTerm`
  - Annahme: Einzusetzender Term hat keine freien Variablen.

α-Äquivalenz

- $t_1 \overset{\alpha}{=} t_2$ — Strukturelle Äquivalenz der Terme t_1 und t_2
- Umformung von t_1 in t_2 allein durch Substitution der (gebundenen) Variablen möglich

α-Äquivalenz

- $t_1 \overset{\alpha}{=} t_2$ — Strukturelle Äquivalenz der Terme t_1 und t_2
- Umformung von t_1 in t_2 allein durch Substitution der (gebundenen) Variablen möglich
- Bspw.:
  - $x \overset{\alpha}{\neq} y$, da x und y frei sind
  - $\lambda x.x \overset{\alpha}{=} \lambda y.y$, durch Umbenennen von x zu y
  - $f\ (\lambda x.y) \overset{\alpha}{=} f\ (\lambda p.y)$
  - $\lambda x.y \overset{\alpha}{\neq} \lambda x.z$

η-Äquivalenz

- $\lambda x.f\ x \overset{\eta}{=} f$, wenn $x \notin fv(f)$
- Wie bei Haskell:

  `all list = foldl (&&) True list` $\Leftrightarrow$

  `all = \list -> foldl (&&) True list` $\Leftrightarrow$

  `all = foldl (&&) True`
- Also:
  - η-Äquivalenz: eher Umformungsschritt als Gleichheitskriterium
  - Formelle Definition von Unterversorgung

β-Reduktion

- Bisher: λ-Terme als (seltsame) Datenstruktur
  Jetzt: Ausführungssemantik

β-Reduktion

- Bisher: λ-Terme als (seltsame) Datenstruktur
  Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ $\Leftrightarrow$
  Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a$

β-Reduktion

- Bisher: λ-Terme als (seltsame) Datenstruktur
  Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ $\Leftrightarrow$
  Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b\,[p \to a]$

β-Reduktion

- Bisher: λ-Terme als (seltsame) Datenstruktur
  Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ $\Leftrightarrow$
  Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b\,[p \to a]$
- „Ausführung“ (besser: Auswertung) von λ-Termen: Anwenden der β-Reduktion, bis Term „konvergiert“
- Term konvergiert $\approx$ Normalform $\approx$ enthält keinen Redex mehr
  - Notation: $t \not\Longrightarrow$

β-Reduktion

- Bisher: λ-Terme als (seltsame) Datenstruktur
  Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“ $\Leftrightarrow$
  Funktionsanwendung $(f\ a)$, mit $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b\,[p \to a]$
- „Ausführung“ (besser: Auswertung) von λ-Termen: Anwenden der β-Reduktion, bis Term „konvergiert“
- Term konvergiert $\approx$ Normalform $\approx$ enthält keinen Redex mehr
  - Notation: $t \not\Longrightarrow$
- $id\ a = (\lambda x.x)\ a \implies x\,[x \to a] = a \not\Longrightarrow$

Beispiel: Church-Booleans

$$
\begin{aligned}
c_{\text{true}} &= \lambda x.\, \lambda y.\, x \\
c_{\text{false}} &= \lambda x.\, \lambda y.\, y \\
\text{AND} &= \lambda a.\, \lambda b.\, a\ b\ c_{\text{false}} \\
&= \lambda a.\, \lambda b.\, (a\ b)\ c_{\text{false}}
\end{aligned}
$$

Funktioniert AND? $\rightsquigarrow$ Wahrheitstabelle aufstellen:

$$
\text{AND}\ c_{\text{true}}\ c_{\text{true}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{true}}\ c_{\text{true}}
$$

Beispiel: Church-Booleans

$$
\begin{aligned}
c_{\text{true}} &= \lambda x.\, \lambda y.\, x \\
c_{\text{false}} &= \lambda x.\, \lambda y.\, y \\
\text{AND} &= \lambda a.\, \lambda b.\, a\ b\ c_{\text{false}} \\
&= \lambda a.\, \lambda b.\, (a\ b)\ c_{\text{false}}
\end{aligned}
$$

Funktioniert AND? $\rightsquigarrow$ Wahrheitstabelle aufstellen:

$$
\begin{aligned}
&\text{AND}\ c_{\text{true}}\ c_{\text{true}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{true}}\ c_{\text{true}} \\
\Rightarrow_\beta\ &(\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{true}}]\ c_{\text{true}} = (\underline{\lambda b.\, c_{\text{true}}\ b\ c_{\text{false}}})\ c_{\text{true}}
\end{aligned}
$$

Beispiel: Church-Booleans

$$
\begin{aligned}
c_{\text{true}} &= \lambda x.\, \lambda y.\, x \\
c_{\text{false}} &= \lambda x.\, \lambda y.\, y \\
\text{AND} &= \lambda a.\, \lambda b.\, a\ b\ c_{\text{false}} \\
&= \lambda a.\, \lambda b.\, (a\ b)\ c_{\text{false}}
\end{aligned}
$$

Funktioniert AND? $\rightsquigarrow$ Wahrheitstabelle aufstellen:

$$
\begin{aligned}
\text{AND}\ c_{\text{true}}\ c_{\text{true}} &= (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{true}}\ c_{\text{true}} \\
\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{true}}]\ c_{\text{true}} &= (\underline{\lambda b.\, c_{\text{true}}\ b\ c_{\text{false}}})\ c_{\text{true}} \\
\Rightarrow_\beta (c_{\text{true}}\ b\ c_{\text{false}})\,[b \to c_{\text{true}}] &= (\underline{\lambda x.\, \lambda y.\, x})\ c_{\text{true}}\ c_{\text{false}}
\end{aligned}
$$

Beispiel: Church-Booleans

$$
\begin{aligned}
c_{\text{true}} &= \lambda x.\, \lambda y.\, x \\
c_{\text{false}} &= \lambda x.\, \lambda y.\, y \\
\text{AND} &= \lambda a.\, \lambda b.\, a\ b\ c_{\text{false}} \\
&= \lambda a.\, \lambda b.\, (a\ b)\ c_{\text{false}}
\end{aligned}
$$

Funktioniert AND? $\rightsquigarrow$ Wahrheitstabelle aufstellen:

$$
\begin{aligned}
&\text{AND}\ c_{\text{true}}\ c_{\text{true}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{true}}\ c_{\text{true}} \\
\Rightarrow_\beta\ &(\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{true}}]\ c_{\text{true}} = (\underline{\lambda b.\, c_{\text{true}}\ b\ c_{\text{false}}})\ c_{\text{true}} \\
\Rightarrow_\beta\ &(c_{\text{true}}\ b\ c_{\text{false}})\,[b \to c_{\text{true}}] = (\underline{\lambda x.\, \lambda y.\, x})\ c_{\text{true}}\ c_{\text{false}} \\
\Rightarrow_\beta\ &(\underline{\lambda y.\, c_{\text{true}}})\ c_{\text{true}} \Rightarrow_\beta c_{\text{true}}\ \checkmark
\end{aligned}
$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{false}}\ t$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{false}}\ t$$

$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{false}}]\ \ t = (\underline{\lambda b.\, c_{\text{false}}\ \ b\ c_{\text{false}}})\ t$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{false}}\ t$$

$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{false}}]\ \ t = (\underline{\lambda b.\, c_{\text{false}}\ b\ c_{\text{false}}})\ t$$

$$\Rightarrow_\beta (c_{\text{true}}\ b\ c_{\text{false}})\,[b \to t] = (\underline{\lambda x.\, \lambda y.\, y})\ t\ c_{\text{false}}$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{false}}\ t$$
$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{false}}]\ t = (\underline{\lambda b.\, c_{\text{false}}\ b\ c_{\text{false}}})\ t$$
$$\Rightarrow_\beta (c_{\text{true}}\ b\ c_{\text{false}})\,[b \to t] = (\underline{\lambda x.\, \lambda y.\, y})\ t\ c_{\text{false}}$$
$$\Rightarrow^2 c_{\text{false}}$$

$$\text{AND } t\ c_{\text{false}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ t\ c_{\text{false}}$$

Beispiel: Church-Booleans

$$\begin{aligned}
&\mathsf{AND}\ c_{\mathsf{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\mathsf{false}}})\ c_{\mathsf{false}}\ t \\
\Rightarrow_\beta\ &(\lambda b.\, a\ b\ c_{\mathsf{false}})\,[a \to c_{\mathsf{false}}]\ \ t = (\underline{\lambda b.\, c_{\mathsf{false}}\ b\ c_{\mathsf{false}}})\ t \\
\Rightarrow_\beta\ &(c_{\mathsf{true}}\ b\ c_{\mathsf{false}})\,[b \to t] = (\underline{\lambda x.\, \lambda y.\, y})\ t\ c_{\mathsf{false}} \\
\Rightarrow^2\ &c_{\mathsf{false}}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{AND}\ t\ c_{\mathsf{false}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\mathsf{false}}})\ t\ c_{\mathsf{false}} \\
\Rightarrow_\beta\ &(\lambda b.\, a\ b\ c_{\mathsf{false}})\,[a \to t]\ \ c_{\mathsf{false}} = (\underline{\lambda b.\, t\ b\ c_{\mathsf{false}}})\ c_{\mathsf{false}}
\end{aligned}$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = \underline{(\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}})}\ c_{\text{false}}\ t$$
$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{false}}]\ t = \underline{(\lambda b.\, c_{\text{false}}\ b\ c_{\text{false}})}\ t$$
$$\Rightarrow_\beta (c_{\text{true}}\ b\ c_{\text{false}})\,[b \to t] = \underline{(\lambda x.\, \lambda y.\, y)}\ t\ c_{\text{false}}$$
$$\Rightarrow^2 c_{\text{false}}$$

$$\text{AND } t\ c_{\text{false}} = \underline{(\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}})}\ t\ c_{\text{false}}$$
$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to t]\ c_{\text{false}} = \underline{(\lambda b.\, t\ b\ c_{\text{false}})}\ c_{\text{false}}$$
$$\Rightarrow_\beta (t\ b\ c_{\text{false}})\,[b \to c_{\text{false}}] = t\ c_{\text{false}}\ c_{\text{false}}$$

Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}}\ t = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ c_{\text{false}}\ t$$
$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to c_{\text{false}}]\ t = (\underline{\lambda b.\, c_{\text{false}}\ b\ c_{\text{false}}})\ t$$
$$\Rightarrow_\beta (c_{\text{true}}\ b\ c_{\text{false}})\,[b \to t] = (\underline{\lambda x.\, \lambda y.\, y})\ t\ c_{\text{false}}$$
$$\Rightarrow^2 c_{\text{false}}$$

$$\text{AND } t\ c_{\text{false}} = (\underline{\lambda a.\, \lambda b.\, a\ b\ c_{\text{false}}})\ t\ c_{\text{false}}$$
$$\Rightarrow_\beta (\lambda b.\, a\ b\ c_{\text{false}})\,[a \to t]\ c_{\text{false}} = (\underline{\lambda b.\, t\ b\ c_{\text{false}}})\ c_{\text{false}}$$
$$\Rightarrow_\beta (t\ b\ c_{\text{false}})\,[b \to c_{\text{false}}] = t\ c_{\text{false}}\ c_{\text{false}}$$
$$\rightsquigarrow c_{\text{false}}$$