

Tutorium 04: λ -Kalkül

Paul Brinkmeier

16. November 2021

Tutorium Programmierparadigmen am KIT

Übungsblatt 3

1 — Streams

```
module Fibs where

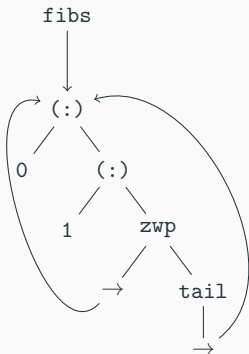
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Auswertung:

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

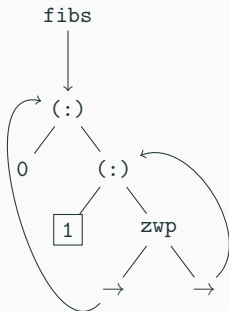
1 — Streams

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



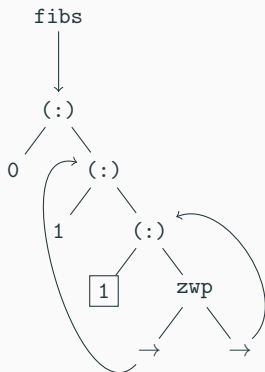
- `zwp = zipWith (+)`
- Laufzeit: Alle Vorkommen von `fibs` beziehen sich auf dasselbe Speicherobjekt (Sharing)
- `tail (x:xs) = xs`

1 — Streams



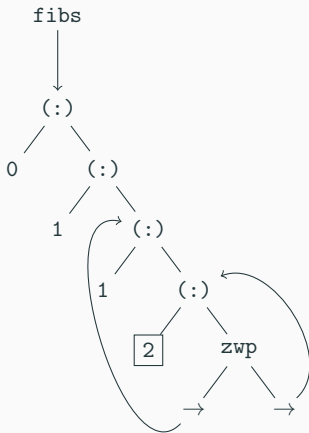
- `tail (x:xs) = xs`
- `fibs !! 1 == 1`
- Keine Berechnung notwendig
- `fibs !! 3?`

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

```
*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

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
```

```
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 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

- „Funktionales Gegenstück zur Turingmaschine“
- Wurde u.a. genutzt um Unlösbarkeit des Halteproblems zu zeigen
- Gibt saftig Punkte in der Klausur
 - 13P. im 19SS
 - 10P. (+15P.) im 18WS
 - 20P. (+15P.) im 18SS

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

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

```
module Lambda where

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

- github.com/pbrinkmeier/pp-tut
- Modul x liegt in demos/x.hs

Begriffe im λ -Kalkül

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

- $fv(t)$ 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:
 - $fv(\lambda x.x) = \emptyset$
 - $fv(\lambda x.y) = \{y\}$

- $fv(t)$ 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`

Substitution

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

Substitution

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

Substitution

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

```
substitute :: (String, Term) -> Term -> Term
```

 - `type Term = LambdaTerm`
 - `fv` braucht ihr dafür nicht

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

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

- $\lambda x.f \ x \stackrel{\eta}{=} f$, 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

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

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

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

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

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

Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

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

Funktioniert AND? \leadsto Wahrheitstabelle aufstellen:

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

Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

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

Funktioniert AND? \leadsto Wahrheitstabelle aufstellen:

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

Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

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

Funktioniert AND? \leadsto Wahrheitstabelle aufstellen:

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

Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

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

Funktioniert AND? \leadsto Wahrheitstabelle aufstellen:

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

Beispiel: Church-Booleans

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

Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{false}} \ t \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{false}}] \ t &= (\lambda b. c_{\text{false}} \ b \ c_{\text{false}}) \ t\end{aligned}$$

Beispiel: Church-Booleans

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

Beispiel: Church-Booleans

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

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

Beispiel: Church-Booleans

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

$$\begin{aligned}\text{AND } t c_{\text{false}} &= (\lambda a. \lambda b. a b c_{\text{false}}) t c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow t] c_{\text{false}} &= (\lambda b. t b c_{\text{false}}) c_{\text{false}}\end{aligned}$$

Beispiel: Church-Booleans

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

$$\begin{aligned}\text{AND } t c_{\text{false}} &= (\lambda a. \lambda b. a b c_{\text{false}}) t c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow t] c_{\text{false}} &= (\lambda b. t b c_{\text{false}}) c_{\text{false}} \\ \Rightarrow_{\beta} (t b c_{\text{false}}) [b \rightarrow c_{\text{false}}] &= t c_{\text{false}} c_{\text{false}}\end{aligned}$$

Beispiel: Church-Booleans

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

$$\begin{aligned}\text{AND } t c_{\text{false}} &= (\lambda a. \lambda b. a b c_{\text{false}}) t c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow t] c_{\text{false}} &= (\lambda b. t b c_{\text{false}}) c_{\text{false}} \\ \Rightarrow_{\beta} (t b c_{\text{false}}) [b \rightarrow c_{\text{false}}] &= t c_{\text{false}} c_{\text{false}} \\ &\leadsto c_{\text{false}}\end{aligned}$$