

PROGRAMMIERUNG

ÜBUNG 6: STRUKTURELLE INDUKTION & λ -KALKÜL (TEIL 1)

Eric Kunze

`eric.kunze@tu-dresden.de`

1. Funktionale Programmierung
 - 1.1 Einführung in Haskell: Listen
 - 1.2 Algebraische Datentypen
 - 1.3 Funktionen höherer Ordnung
 - 1.4 Typpolymorphie & Unifikation
 - 1.5 Beweis von Programmeigenschaften
 - 1.6 λ - **Kalkül**
2. Logikprogrammierung
3. Implementierung einer imperativen Programmiersprache
4. Verifikation von Programmeigenschaften
5. H_0 – ein einfacher Kern von Haskell

Induktionsbeweise

Aufgabe 1

VOLLSTÄNDIGE INDUKTION AUF \mathbb{N}

Definition: natürliche Zahlen $\mathbb{N} := \{0, 1, \dots\}$

Basisfall: $0 \in \mathbb{N}$

Rekursionsfall: $x + 1 \in \mathbb{N}$ für $x \in \mathbb{N}$

Beweis von Eigenschaften: Eigenschaft = Prädikat P

zu zeigen: für alle $x \in \mathbb{N}$ gilt $P(x)$
--

vollständige Induktion:

- ▶ **Induktionsanfang:**

zeige $P(x)$ für $x = 0$

- ▶ **Induktionsvoraussetzung:**

Sei $x \in \mathbb{N}$, sodass $P(x)$ gilt.

$P(x)$ gilt noch nicht für *alle* $x \in \mathbb{N}$

- ▶ **Induktionsschritt:**

zeige $P(x + 1)$ unter Nutzung der Induktionsvoraussetzung

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen xs

Basisfall: $xs = []$

Rekursionsfall: $xs = (y:ys)$ für $ys :: [a]$

Beweis von Programmeigenschaften: Eigenschaft = Prädikat P

zu zeigen: für alle $xs :: [a]$ gilt $P(xs)$
--

Induktion auf Listen:

- ▶ **Induktionsanfang:**
zeige $P(xs)$ für $xs == []$
- ▶ **Induktionsvoraussetzung:**
Sei $xs :: [a]$ eine Liste für die $P(xs)$ gilt.
- ▶ **Induktionsschritt:**
zeige $P(x:xs)$ für alle $x :: a$ unter Nutzung der Induktionsvoraussetzung

<i>Allgemeiner Hinweis:</i> Es müssen immer alle Variablen quantifiziert werden!

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: Nil oder Leaf x für $x :: a$

Rekursionsfall: Branch x l r für $x :: a$ und $l, r :: \text{BinTree } a$

zu zeigen: für alle $t :: \text{BinTree } a$ gilt $P(t)$
--

strukturelle Induktion:

► **Induktionsanfang:**

zeige $P(t)$ für $t == \text{Nil}$ oder $t == \text{Leaf } x$ für alle $x :: a$

► **Induktionsvoraussetzung:**

Seien $l, r :: \text{BinTree } a$ zwei Bäume, sodass $P(l)$ und $P(r)$ gilt.

► **Induktionsschritt:**

zeige $P(\text{Branch } x \ l \ r)$ für alle $x :: a$ unter Nutzung der Induktionsvoraussetzung

<i>Allgemeiner Hinweis:</i> Es müssen immer alle Variablen quantifiziert werden!

AUFGABE 1

```
1 data Tree a = Node a (Tree a) (Tree a) | Leaf a
2
3 mirror :: Tree a -> Tree a
4 mirror (Node x t1 t2) = Node x (mirror t2) (mirror t1)
5 mirror (Leaf x) = Leaf x
6
7 yield :: Tree a -> [a]
8 yield (Node _ t1 t2) = yield t1 ++ yield t2
9 yield (Leaf x) = [x]
```

verwendbare Eigenschaften:

$$\text{reverse } [x] = [x] \quad (\text{E1})$$

$$\text{reverse } (xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs \quad (\text{E2})$$

Mittels struktureller Induktion ist folgende Aussage zu zeigen:

Für jeden Typ a und jeden Baum $t :: \text{Tree } a$ gilt

$$\text{reverse } (\text{yield } t) = \text{yield } (\text{mirror } t)$$

AUFGABE 1

Induktionsanfang: Sei a ein Typ und $x :: a$ beliebig sowie $t = \text{Leaf } x$.

linke Seite: $\text{reverse } (\text{yield } (\text{Leaf } x)) \stackrel{(9)}{=} \text{reverse } [x] \stackrel{(E1)}{=} [x]$

rechte Seite: $\text{yield } (\text{mirror } (\text{Leaf } x)) \stackrel{(5)}{=} \text{yield } (\text{Leaf } x) \stackrel{(9)}{=} [x]$

Induktionsvoraussetzung: Seien a ein Typ und $l, r :: \text{Tree } a$, sodass

$\text{reverse } (\text{yield } r) = \text{yield } (\text{mirror } l)$ (IV1)

$\text{reverse } (\text{yield } r) = \text{yield } (\text{mirror } r)$ (IV2)

gilt.

AUFGABE 1

Induktionsschritt: Sei a ein Typ und $x :: a$ beliebig. Es gilt

```
reverse (yield (Node x l r))  
   $\stackrel{(8)}{=}$  reverse (yield l ++ yield r)  
   $\stackrel{(E2)}{=}$  reverse (yield r) ++ reverse (yield l)  
   $\stackrel{(IV1)}{=}$  reverse (yield r) ++ yield (mirror l)  
   $\stackrel{(IV2)}{=}$  yield (mirror r) ++ yield (mirror l)  
   $\stackrel{(8)}{=}$  yield (Node x (mirror r) (mirror l))  
   $\stackrel{(4)}{=}$  yield (mirror (Node x l r))
```

Der λ -Kalkül

Aufgaben 2 & 3

- ▶ weitere funktionale Programmiersprache
- ▶ Programme = λ -Terme
- ▶ Vorstellung: *anonyme* Funktionen

Sei X eine Menge mit Variablen, Σ eine Menge mit Symbolen. Die gültigen λ -Terme sind *induktiv* definiert:

1. **Atome** (Variablen oder Symbole) sind gültige λ -Terme.
2. **Abstraktion**: Ist t ein gültiger λ -Term und $x \in X$ eine Variable, dann ist auch $(\lambda x. t)$ ein gültiger λ -Term.
3. **Applikation**: Sind t_1 und t_2 gültige λ -Terme, dann ist auch $(t_1 \ t_2)$ ein gültiger λ -Term.

BEISPIELE (INFORMELL)

Vorstellung: Jeder λ -Term beschreibt eine anonyme Funktion.

- Abstraktion gibt das Argument an:

$$\lambda x. t \quad \leftrightarrow \quad f(x) = t$$

- Applikation beschreibt Funktionsanwendung (Einsetzen):

$$((\lambda x. t) 2) \quad \leftrightarrow \quad f(2)$$

Beispiel:

$$\begin{aligned} \text{quadriere} &= \lambda x. x \cdot x \\ ((\lambda x. x \cdot x) 2) &= 2 \cdot 2 = 4 \end{aligned}$$

↗ β -Reduktion

- Applikation ist linksassoziativ:

$$((t_1 \ t_2) \ t_3) = t_1 \ t_2 \ t_3$$

- mehrfache Abstraktion:

$$(\lambda x_1. (\lambda x_2. (\lambda x_3. t))) = \lambda x_1 x_2 x_3. t$$

- Applikation vor Abstraktion:

$$\begin{aligned} (\lambda x. x \ y) &= (\lambda x. (x \ y)) \\ &\neq ((\lambda x. x) \ y) \end{aligned}$$

Mengen $FV(t)$ und $GV(t)$ geben frei bzw. gebunden *vorkommende* Variablen von t an — induktive Definition

- ▶ einzelne **Variablen** sind immer frei:
 $x \in X \Rightarrow FV(x) = \{x\}, GV(x) = \emptyset$
- ▶ **Symbole** sind weder frei noch gebunden
- ▶ **Applikation:** Sei $t = (t_1 \ t_2)$. Dann
 $\Rightarrow FV(t) = FV(t_1) \cup FV(t_2), \quad GV(t) = GV(t_1) \cup GV(t_2)$
- ▶ **Abstraktion:** $t = \lambda x. t'$
 $\Rightarrow FV(t) = FV(t') \setminus \{x\}, \quad GV(t) = GV(t') \cup \{x\}$

β -Reduktion

Seien $s, t \in \lambda(\Sigma)$ gültige λ -Terme und es gilt $GV(t) \cap FV(s) = \emptyset$.

$$(\lambda x. t) s \longrightarrow_{\beta} t[x/s]$$

- Bedeutung von $t[x/s]$: Ersetze jedes *freie* Vorkommen von x in t durch s .
- Erinnerung: Vorstellung der Applikation als „Einsetzen“ in Funktionen
- beachte: Abstraktion λx entfällt

Bsp.: Seien die Symbole gegeben durch $\Sigma = \{3, a\}$.

$$(\lambda x. \underbrace{+x3}_{GV=\emptyset}) (\underbrace{\lambda z. a}_{FV=\emptyset}) \longrightarrow_{\beta} + (\lambda z. a)3$$

α – KONVERSION

- ▶ Was machen wir, wenn Voraussetzung $FV(t) \cap GV(t) = \emptyset$ für β -Reduktion nicht erfüllt ist?
- ▶ einfacher Ausweg: entsprechende Variablen umbenennen, sodass Bedingung erfüllt ist

α -Konversion

Sei $t \in \lambda(\Sigma)$ und $z \notin GV(t) \cup FV(t)$.

$$(\lambda x. t) \longrightarrow_{\alpha} \lambda z. t[x/z]$$

Bsp.: Seien die Symbole gegeben durch $\Sigma = \{3, a\}$.

$$(\lambda x. (\underbrace{\lambda y. + xy}_{GV=\{y\}})) (\underbrace{y}_{FV=\{y\}}) \longrightarrow_{\alpha} (\lambda x. (\underbrace{\lambda z. + xz}_{GV=\{z\}})) (\underbrace{y}_{FV=\{y\}})$$

Nutzt man sowohl α -Konversionen \Rightarrow_α als auch β -Reduktionen \Rightarrow_β , so spricht man von der *Rechenvorschrift* \Rightarrow des λ -Kalküls.

Führt man mehrere Schritte direkt aus, so schreibt man \Rightarrow^* statt \Rightarrow .

Wenn $t \Rightarrow^* s$ und es gibt keine λ -Terme s_1, s_2 mit $s \Rightarrow_\alpha^* s_1 \Rightarrow_\beta^* s_2$, dann heißt s (β -)**Normalform** von t .

Anschauung:

- ▶ möglichst einfache Form eines Lambda-Terms
- ▶ „fertig ausgerechnete“ Funktion

Die Rechenvorschrift \Rightarrow ist *konfluent*, d. h. für alle λ -Terme t, t_1, t_2 gilt: wenn $t \Rightarrow^* t_1$ und $t \Rightarrow^* t_2$, dann gibt es einen λ -Term s mit $t_1 \Rightarrow^* s$ und $t_2 \Rightarrow^* s$.

Damit gilt auch: wenn eine Normalform existiert, dann ist sie *eindeutig* (egal welche Schritte man zwischendurch auswählt).

Vorgehen:

1. Bestimmung von Stellen, an denen reduziert werden kann

Anforderungen: (Teil-)Term der Form $(\underbrace{(\lambda x. t)}_{=t'} s)$

- ▷ Applikation $(t' s)$: es muss ein Term s zum Einsetzen vorhanden sein
- ▷ Abstraktion in $t' = (\lambda x. t)$: der erste Term braucht eine „Variable“ x , für die etwas eingesetzt werden kann

Achtung: implizite Klammerung (Linksassoziativität) beachten!

2. Bestimmung gebundener und freier Vorkommen: $((\lambda x. \underbrace{t}_{\text{GV}}) \underbrace{s}_{\text{FV}})$

3. Falls $\text{GV}(t) \cap \text{FV}(s) \neq \emptyset$: α -Konversion

- ▷ Umbenennung der gebunden Vorkommen in t

4. Falls $\text{GV}(t) \cap \text{FV}(s) = \emptyset$: β -Reduktion

- ▷ Streichen der Abstraktion λx .
- ▷ Setze für jedes Vorkommen von x in t den Term s sein

AUFGABE 2 — TEIL (A)

- ▶ $t_1 = (\lambda x. x y) (\lambda y. y)$:
 - ▷ $GV(t_1) = \{x, y\}$
 - ▷ $FV(t_1) = \{y\}$
- ▶ $t_2 = (\lambda x. (\lambda y. z (\lambda z. z (\lambda x. y))))$
 - ▷ $GV(t_2) = \{x, y, z\}$
 - ▷ $FV(t_2) = \{z\}$
- ▶ $t_3 = (\lambda x. (\lambda y. z (y z))) (\lambda x. y (\lambda y. y))$
 - ▷ $GV(t_3) = \{x, y\}$
 - ▷ $FV(t_3) = \{y, z\}$

AUFGABE 2 — TEIL (B)

Term 1

$$\begin{aligned} & (\lambda x. \underbrace{(\lambda y. x \ z \ (y \ z))}_{GV=\{y\}}) \underbrace{(\lambda x. y \ (\lambda y. y))}_{FV=\{y\}} \\ \Rightarrow_{\alpha} & (\lambda x. \underbrace{(\lambda y_1. x \ z \ (y_1 \ z))}_{GV=\{y\}}) \underbrace{(\lambda x. y \ (\lambda y. y))}_{FV=\{y\}} \\ \Rightarrow_{\beta} & (\lambda y_1. \underbrace{(\lambda x. y \ (\lambda y. y))}_{GV=\{y\}}) \underbrace{z}_{FV=\{z\}} (y_1 \ z)) \\ \Rightarrow_{\beta} & (\lambda y_1. (y \ (\lambda y. y)) (y_1 \ z)) \\ & = (\lambda y_1. y \ (\lambda y. y) (y_1 \ z)) \end{aligned}$$

AUFGABE 2 — TEIL (B)

Term 2

$$\begin{aligned} & (\lambda x. (\underbrace{\lambda y. (\lambda z. z)}_{GV=\{y,z\}})) \underbrace{x}_{FV=\{x\}} (+ y 1) \\ \Rightarrow_{\beta} & (\lambda y. (\underbrace{\lambda z. z}_{GV=\{z\}})) \underbrace{(+ y 1)}_{FV=\{y\}} \\ \Rightarrow_{\beta} & (\lambda z. z) \end{aligned}$$

AUFGABE 2 — TEIL (B)

Term 3

$$\begin{aligned}
 & (\lambda x. (\lambda y. x (\lambda z. y z))) (((\lambda x. (\lambda y. y)) \underbrace{8}_{\substack{GV=\{y\} \\ FV=\emptyset}}) (\lambda x. (\lambda y. y) x)) \\
 \Rightarrow_{\beta} & (\lambda x. (\lambda y. x (\lambda z. y z))) ((\lambda y. y)) (\lambda x. (\lambda y. y) x) \\
 \Rightarrow_{\beta} & (\lambda x. (\lambda y. x (\lambda z. y z))) ((\lambda y. y) (\lambda x. (\lambda y. \underbrace{y}_{\substack{GV=\emptyset}}) \underbrace{x}_{\{x\}})) \\
 \Rightarrow_{\beta} & (\lambda x. (\lambda y. x (\lambda z. y z))) ((\lambda y. \underbrace{y}_{\substack{GV=\emptyset}}) (\lambda x. \underbrace{x}_{\substack{FV=\emptyset}})) \\
 \Rightarrow_{\beta} & (\lambda x. (\lambda y. x (\lambda z. y z))) (\lambda x. x) \\
 \Rightarrow_{\beta} & (\lambda y. (\lambda x. \underbrace{x}_{\substack{GV=\emptyset}}) (\lambda z. y z)) \\
 \Rightarrow_{\beta} & (\lambda y. (\lambda z. y z)) = (\lambda y z. y z)
 \end{aligned}$$

AUFGABE 2 — TEIL (B)

Term 4

$$(\lambda h. (\lambda x. h (x x)) (\lambda x. h (x x))) ((\lambda x. \underbrace{x}_{GV=\emptyset}) \underbrace{(+ 1 5)}_{FV=\emptyset})$$

$$\Rightarrow_{\beta} (\lambda h. (\lambda x. \underbrace{h (x x)}_{GV=\emptyset}) \underbrace{(\lambda x. h (x x))}_{FV=\{h\}}) (+ 1 5)$$

$$\Rightarrow_{\beta} (\lambda h. h ((\lambda x. \underbrace{h (x x)}_{GV=\emptyset}) \underbrace{(\lambda x. h (x x))}_{FV=\{h\}})) (+ 1 5)$$

$$\Rightarrow_{\beta} (\lambda h. h (h ((\lambda x. \underbrace{h (x x)}_{GV=\emptyset}) \underbrace{(\lambda x. h (x x))}_{FV=\{h\}}))) (+ 1 5)$$

→ endlose Rekursion, bei der h durch $(+ 1 5)$ noch reduziert werden könnte

$$\Rightarrow_{\beta} (+ 1 5) ((+ 1 5) ((\lambda x. (+ 1 5) (x x)) (\lambda x. (+ 1 5) (x x))))$$

AUFGABE 2 — TEIL (B)

Term 5

$$\begin{aligned} & (\lambda f. (\underbrace{\lambda a. (\lambda b. f \ a \ b)}_{GV=\{a,b\}}) (\underbrace{\lambda x. (\lambda y. x)}_{FV=\emptyset})) \\ \Rightarrow_{\beta} & (\lambda a. (\lambda b. (\lambda x. (\underbrace{\lambda y. x}_{GV=\{y\}}) \underbrace{a}_{FV=\{a\}} \ b))) \\ \Rightarrow_{\beta} & (\lambda a. (\lambda b. (\lambda y. (\underbrace{a}_{GV=\emptyset} \ \underbrace{b}_{FV=\{b\}})))) \\ \Rightarrow_{\beta} & (\lambda a. (\lambda b. a)) \\ = & (\lambda ab. a) \end{aligned}$$

AUFGABE 3

(a) A mit $A \ t \ s \ u \Rightarrow^* s$: $A = (\lambda x y z . y)$

(b) B mit $B \ t \ s \Rightarrow^* s \ t$: $B = (\lambda x y . y x)$

(c) C mit $C \ C \Rightarrow^* C \ C$: $C = (\lambda x . x x)$

denn: $(\lambda x . \underbrace{xx}_{GV=\emptyset}) (\underbrace{\lambda x . xx}_{FV=\emptyset}) \Rightarrow^\beta (\lambda x . xx)(\lambda x . xx)$

(d) D mit $D \Rightarrow^* D$: $D = (C \ C)$

(e) E mit $E \ E \ t \Rightarrow^* E \ t \ E$: $E = (\lambda x y . x y x)$

denn:

$$\begin{aligned} (\lambda x \underbrace{y . x y x}_{GV=\{y\}}) (\underbrace{\lambda x y . x y x}_{FV=\emptyset}) \ t &\Rightarrow^\beta (\lambda y . (\lambda x y . x y x) \ y \ (\lambda x y . x y x)) \ t \\ &\Rightarrow^\beta \underbrace{(\lambda x y . x y x)}_{=E} \ t \ \underbrace{(\lambda x y . x y x)}_{=E} \end{aligned}$$