Programmierparadigmen

Kapitel 3b
Lambda Kalkül

(Dieses Kapitel orientiert sich an Material von Prof. Dr.-Ing. Snelting, KIT)

Kalküle

Kalküle sind

- Minimalistische Programmiersprachen zur Beschreibung von Berechnungen,
- mathematische Objekte, über die Beweise geführt werden können

In dieser Vorlesung:

- λ-Kalkül (Church, Landin) für sequentielle (funktionale / imperative) Sprachen

Beispiele weiterer Kalküle:

- CSP (Hoare) *Communicating Sequential Processes* – für nebenläufige Programme mit Nachrichtenaustausch
- *π*-Kalkül (Milner) für nebenläufige, *mobile* Programme

Der untypisierte λ-Kalkül

- Turing-mächtiges Modell funktionaler Programme
- Auch: Beschreibung sequentieller imperativer Konstrukte

Definition der λ-Terme:

Die Klasse Λ der Lambda-Terme ist die kleinste Klasse, welche die folgenden Eigenschaften erfüllt:

- Wenn x eine Variable ist, dann ist $x \in \Lambda$
- Wenn $M \in \Lambda$ ist, dann ist $(\lambda x M) \in \Lambda$ (Abstraktion)
- Wenn $M, N \in \Lambda$ sind, dann ist $(MN) \in \Lambda$ (Funktionsanwendung)

- Um Klammern "einzusparen" verwendet man oft eine alternative Notation: λx.M
- Bei mehreren zu bindenden Variablen: λxyz.M = (λx(λy(λzM)))

Strukturelle Induktion

- Aufgrund des „rekursiven" Aufbaus der Definition der Klasse Λ der Lambda-Terme, können Aussagen über Lambda-Terme mittels **"struktureller Induktion"** geführt werden:
  - Hierbei folgt der Induktionsbeweis der Struktur der Lambda-Terme, wie er in der Definition vorgegeben wird
- Beispiel: Jeder Term in Λ ist wohlgeklammert
  - Induktionsanfang: trivial, da jede Variable ein wohlgeklammerter Lambda-Term ist.
  - Induktionsannahme: M, N sind wohlgeklammerte Lambda-Terme
  - Induktionsschritt: dann sind auch die Terme (MN) und (λxM) wohlgeklammert.

Der untypisierte λ-Kalkül

λ-Terme

Bezeichnung	**Notation**	**Beispiele**	
Variablen	x	x	y
Abstraktion	λx. t	λy.*0*	λf. λx. λy.fyx
Funktionsanwendung	$t_1\ t_2$	f 42	(λx. x + 5) 7

(weitere primitive Operationen nach Bedarf) 17, True, +, · , . . .

Variablenkonvention:

x, y, f	sind konkrete Programmvariablen
x, y, z	sind Meta-Variablen für Programmvariablen
t, t´, t_1, t_2, ...	bezeichnen immer einen -Term

Funktionsanwendung ist linksassoziativ und bindet stärker als Abstraktion

λx.fxy = λx. *((*fx) y)

Abstraktion ist rechtsassoziativ:

λx.λy.fxy = *(*λx.*(*λy.fxy*))*

Variablenbindung bei Abstraktion

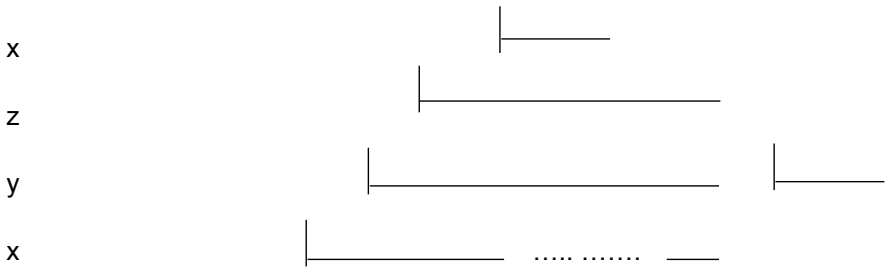
Variablenbindung in Haskell (erlaubt anonyme Lambda-Funktionen):
Anonyme Funktion: (\ x -> x + y) 5
let-Ausdruck: **let** x = 5 **in** x + y

Analog bei λ-Abstraktionen: λx. t bindet die Variable x im Ausdruck t
Beispiele:

- λx. λy. f y x bindet x in λy. f y x, das selbst y in f y x bindet.
- f ist frei in λx. λy. f y x.

Innere Abstraktionen können äußere Variablen verdecken:

(λx. λy. λz. f (λx. z + x) (y x)) (λy. y+ x)



Freie und gebundene Variablen

- Die Menge der **freien Variablen** eines Terms M wird mit FV(M) bezeichnet und ist wie folgt induktiv definiert:
  - FV(x) = {x}
  - FV(MN) = FV(M) ∪ FV(N)
  - FV(λx.M) = FV(M) – {x}
- Übung:
  Definieren Sie analog die Menge der gebundenen Variablen GV(M)
- Ein Lambda-Term ohne freie Variablen heißt **Kombinator**
- Einige besonders wichtige Kombinatoren haben eigene Namen:
  - Identitätsfunktion: $I \equiv \lambda x.x$
  - Konstanten-Funktional: $K \equiv \lambda xy.x$
  - Fixpunkt-Kombinator: $Y \equiv \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
    (dieser wird später erklärt)

α - Äquivalenz

Namen gebundener Variablen
- dienen letztlich nur der Dokumentation
- entscheidend sind die Bindungen

α - Äquivalenz
t_1 und t_2 heißen α-äquivalent ($t_1 \overset{\alpha}{=} t_2$), wenn t_1 in t_2 durch konsistente Umbenennung der λ-gebundenen Variablen überführt werden kann.

Beispiele:

$$\lambda x.\ x \overset{\alpha}{=} \lambda y.\ y$$
$$\lambda x.\ \lambda z.\ f(\lambda y.\ zy)\ x \overset{\alpha}{=} \lambda y.\ \lambda x.\ f(\lambda\ z.\ xz)\ y$$

aber

$$\lambda x.\ \lambda z.\ f(\lambda y.\ zy)\ x \overset{\alpha}{\neq} \lambda x.\ \lambda z.\ g(\lambda y.\ zy)\ x$$
$$\lambda z.\ \lambda z.\ f(\lambda y.\ zy)\ z \overset{\alpha}{\neq} \lambda x.\ \lambda z.\ f(\lambda y.\ zy)\ x$$

η - Äquivalenz

Extensionalitäts-Prinzip:

- Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

η-Äquivalenz

Terme λx. f x und f heißen η-äquivalent ($\lambda x.\ f\ x \overset{\eta}{=} f$) falls x nicht freie Variable von f ist

Beispiele:

$$\lambda x.\lambda y.\ \underline{f\ z\ x}\ y \overset{\eta}{=} \lambda x.\ f\ z\ x$$
$$f\ z \overset{\eta}{=} \lambda x.\ \underline{f\ z}\ x$$
$$\lambda x.\ x \overset{\eta}{=} \lambda x.\ \underline{(\lambda x.\ x)}\ x$$

Aber

$$\lambda x.\ \underline{f\ x}\ x \overset{\eta}{\neq} f\ x$$

Ausführung von λ-Termen

Redex	Ein λ-Term der Form $(\lambda x.\ t_1)\ t_2$ heißt Redex.
β-Reduktion	β-Reduktion entspricht der Ausführung der Funktionsanwendung auf einem Redex: $(\lambda x.\ t_1)\ t_2 \Rightarrow t_1\ [x \rightarrow t_2]$
Substitution	$t_1\ [x \rightarrow t_2]$ erhält man aus dem Term t_1, wenn man alle *freien* Vorkommen von x durch t_2 ersetzt.
Normalform	Ein Term, der nicht weiter reduziert werden kann, heißt in Normalform.

Beispiele:

$$(\lambda x.\ x)\ y \Rightarrow x\ [x \rightarrow y] = y$$
$$(\lambda x.\ x\ (\lambda x.\ x))\ (y\ z) \Rightarrow (x\ (\lambda x.\ x))\ [x \rightarrow y\ z] = (y\ z)\ (\lambda x.\ x)$$

Auswertungsstrategien (1)

Wenn es in einem Term mehrere Redexe gibt, welchen reduziert man?

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

Auswertungsstrategien (2)

Wenn es in einem Term mehrere Redexe gibt, welchen reduziert man?

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

Volle β-Reduktion Jeder Redex kann jederzeit reduziert werden

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

$$\Rightarrow\ (\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ z))$$

$$\Rightarrow\ (\lambda x.\ x)\ (\lambda z.\ z)$$

$$\Rightarrow\ \lambda z.\ z \quad \not\Rightarrow$$

Auswertungsstrategien (3)

Wenn es in einem Term mehrere Redexe gibt, welchen reduziert man?

$$(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$

Volle β-Reduktion	Jeder Redex kann jederzeit reduziert werden
Normalreihenfolge	Immer der linkeste äußerste Redex wird reduziert

$(\underline{\lambda x.\ x})\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$

$\Rightarrow\ (\underline{\lambda x.\ x})\ (\lambda z.\ (\lambda x.\ x)\ z)$

$\Rightarrow\ \lambda z.\ (\underline{\lambda x.\ x})\ z$

$\Rightarrow\ \lambda z.\ z \quad \not\Rightarrow$

Braucht man primitive Operationen?

Nicht unbedingt – Kodierung mit Funktionen höherer Ordnung:

Beispiel: *let*

	let $x = t_1$ *in* t_2	wird zu	$(\lambda x.\ t_2)\ t_1$
Beispiel:	*let* $x = g\ y$ *in* $f\ x$	berechnet	$f\ (g\ y)$
	$(\underline{\lambda x.\ f\ x})\ (g\ y)$	$\Rightarrow$	$f\ (g\ y)$

Kodierung boolescher Werte

Church-Booleans

True	wird zu	C_{true}	=	λt. λf. t
False	wird zu	C_{false}	=	λt. λf. f
If-then-else	wird zu	If	=	λa. a

- *if True then x else y* ergibt:
  (λa. a) (λt. λf. t) x y ⇒ (λt. λf. t) x y ⇒ (λf. x) y ⇒ x
- b_1 && b_2 ist äquivalent zu *if* b_1 *then* b_2 *else False*

⇒ b_1 && b_2 wird zu (λa. a) b_1 b_2 C_{false}

⇒ b_1 && b_2 wird zu (λa. a) b_1 b_2 (λt. λf. f)

- True && True ergibt:
  (λa. a) C_{true} C_{true} (λt. λf. f)

⇒ (λt. λf. t) (λt. λ f. t) (λt. λf. f)

⇒ (λf. (λt. λ f. t)) (λt. λf. f) ⇒ λt. λf.t = C_{true}

Kodierung natürlicher Zahlen

Church – Zahlen

Eine (natürliche) Zahl drückt aus, wie oft etwas (s) geschehen soll.

c_0 = λs. λz. z

c_1 = λs. λz. s z

c_2 = λs. λz. s (s z)

c_3 = λs. λz. s (s(s z))

…

c_n= λs. λz. s^n z

Nachfolgefunktion:

succ = λn. λs. λz. s (n s z)

n Church – Zahl,

d.h. von der Form λs. λz....

succ(c_2) = (λn. λs. λz. s (n s z)) (λs. λz. s (s z))

⇒ λs. λz. s ((λs. λz. s (s z)) s z)

⇒ λs. λz. s ((λz. s (s z)) z)

⇒ λs. λz. s (s (s z)) = c_3

Rechnen mit Church – Zahlen (1)

Arithmetische Operationen

Addition: plus = $\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$

Multiplikation: times = $\lambda m.\ \lambda n.\ \lambda s.\ n\ (m\ s)$

$\overset{\eta}{=}\ \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ (m\ s)\ z$

Exponentiation: exp = $\lambda m.\ \lambda n.\ n\ m$

$\overset{\eta}{=}\ \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ m\ s\ z$

$\text{plus}\ c_2\ c_3 = (\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z))\ c_2\ c_3$

$\overset{2}{\Rightarrow}\ \lambda s.\ \lambda z.\ c_2\ s\ (c_3\ s\ z))$

$\overset{2}{\Rightarrow}\ \lambda s.\ \lambda z.\ (\lambda s.\ \lambda z.\ s\ (s\ z))\ s\ ((\lambda s.\ \lambda z.\ s\ (s\ (s\ z)))\ s\ z)$

$\Rightarrow\ \lambda s.\ \lambda z.\ (\lambda z.\ s\ (s\ z))\ ((\lambda s.\ \lambda z.\ s\ (s\ (s\ z)))\ s\ z)$

$\Rightarrow\ \lambda s.\ \lambda z.\ s\ (s\ ((\lambda s.\ \lambda z.\ s\ (s\ (s\ z)))\ s\ z)$

$\Rightarrow\ \lambda s.\ \lambda z.\ s\ (s\ ((\lambda z.\ s\ (s\ (s\ z)))\ z)$

$\Rightarrow\ \lambda s.\ \lambda z.\ s\ (s\ (\ s\ (\ s\ (s\ z)))) = c_5$

Rechnen mit Church – Zahlen (2)

Arithmetische Operationen

Addition: plus = $\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$

Multiplikation: times = $\lambda m.\ \lambda n.\ \lambda s.\ n\ (m\ s)$

$\overset{\eta}{=}\ \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ (m\ s)\ z$

Exponentiation: exp = $\lambda m.\ \lambda n.\ n\ m$

$\overset{\eta}{=}\ \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ m\ s\ z$

Idee zu exp:

$\text{exp}\ c_m\ c_n \overset{2}{\Rightarrow} c_n\ c_m \Rightarrow (\lambda s.\ \lambda z.\ s^n\ z)\ (\lambda s.\ \lambda z.\ s^m\ z)$

$\Rightarrow\ \lambda z.\ (\lambda s.\ \lambda z.\ s^m\ z)^n\ z$

(per Induktion über n) $\overset{\alpha\ \beta\ \eta}{\Rightarrow}\ \lambda s.\ \lambda z.\ \lambda z.\ s^{m^n}\ z = c_{m^n}$

Rechnen mit Church – Zahlen (3)

Arithmetische Operationen

Vorgänger: pred = λn.λs.λx. n (λy.λz. z (y s))(K x) I

Subtraktion: sub = λn.λm. m pred n

Nullvergleich: isZero = λn. n (λx. C_{false}) C_{true}

isZero(c_0) = (λn. n (λx. C_{false}) C_{true}) (λs. λz. z)

⇒ (λs. λz. z) (λx. C_{false}) C_{true}

⇒ (λz. z) C_{true} ⇒ C_{true}

isZero(c_1) = (λn. n (λx. C_{false}) C_{true}) (λs. λz. s z)

⇒ (λs. λz. s z) (λx. C_{false}) C_{true}

⇒ (λz. (λx. C_{false}) z) C_{true}

⇒ (λx. C_{false}) C_{true} ⇒ C_{false}

(Bemerkung: I und K sind die Identitätsfunktion bzw. das Konstanten-Funktional)

Rechnen mit Church – Zahlen (4)

pred(c_2) = (λn.λs.λx. n (λy.λz. z (y s))(K x) I) (λs. λz. s (s z))

⇒ λs.λx. (λs´. λz´. s´ (s´ z´)) (λy.λz. z (y s)) (K x) I

⇒ λs.λx. (λz´. (λy.λz. z (y s)) ((λy.λz. z (y s)) z´)) (K x) I

⇒ λs.λx. (λy.λz. z (y s)) ((λy.λz. z (y s)) (K x)) I

⇒ λs.λx. (λz´. z´ (((λy.λz. z (y s)) (K x)) s)) I

⇒ λs.λx. I (((λy.λz. z (y s)) (K x)) s)

⇒ λs.λx. I ((λz. z (K x s)) s)

⇒ λs.λx. I (s (K x s))

⇒ λs.λx. (λx´.x´) (s ((λx´´.λy.x´´) x s))

⇒ λs.λx. s ((λx´´.λy.x´´) x s)

⇒ λs.λx. s ((λy.x) s)

⇒ λs.λx. s x = c_1

Divergenz

Bisherige Beispiele werten zu einer Normalform aus.

Aber:

$$\omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow \ldots$$

λx. x x wendet sein Argument auf das Argument selbst an

⇒ dadurch reproduziert ω sich selbst.

Divergenz

Terme, die nicht zu einer Normalform auswerten, divergieren.

Diese modellieren unendliche Ausführungen.

Der Fixpunktsatz

Fixpunktsatz

Für alle F ϵ Λ existiert ein X ϵ Λ so dass gilt: F X = X

- Der Fixpunktsatz besagt, dass im Lambda-Kalkül jeder Term einen Fixpunkt hat, d.h. einen Wert, der auf sich selber abgebildet wird.
- Beweis:
  - Zu einem beliebigen F sei W = λx.F(x x) und X = (W W)
  - Dann gilt: X ≡ W W ≡ (λx.F(x x)) W ≡ F(W W) ≡ F X
- Bemerkungen:
  - Für einige Lambda-Terme ist die Identifikation eines Fixpunktes einfach, z.B. für den Term λx.x (alle Terme sind Fixpunkte)
  - Für andere Terme, wie z.B. λxy.x y (= λx.λy.x y) ist das nicht so klar
  - Der Beweis des Fixpunktsatzes ist konstruktiv, d.h. er liefert zu jedem Lambda-Term einen Fixpunkt

Anwendung des Fixpunktsatzes

- Aufgabe: Berechne den Fixpunkt zu dem Term λxy.x y
  - Lösungsansatz: W ≡ λx.(λxy.x y)(x x) ≡ λx.λy.(x x) y ≡ λxy.(x x) y
  - Damit ist der gesuchte Fixpunkt X ≡ ((λxy.(x x) y) (λxy.(x x) y))
  - Nachrechnen:

    (λxy. x y) ((λxy.(x x) y) (λxy.(x x) y))

    ≡ (λx.λy. x y) ((λxy.(x x) y) (λxy.(x x) y))

    ≡ λy.((λxy.(x x) y) (λxy.(x x) y)) y

    ≡ (λxy.(x x) y) (λxy.(x x) y)

    ≡ X
- Bemerkung: Der so für die Identitätsfunktion λx.x konstruierte Fixpunkt ist übrigens (λx.x x) (λx.x x), er spielt die besondere Rolle des Standardterms ⊥ für nicht-terminierende Ausführungen

Der Fixpunkt-Kombinator

Im Ergebnis unserer Diskussion des Fixpunktsatzes definieren wir den **Fixpunkt-Kombinator** wie folgt:

$$Y \equiv \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

- Dieser Kombinator spielt eine wichtige Rolle bei der Definition rekursiver Funktionen im Lambda-Kalkül, wie wir im folgenden sehen werden
- Für jeden Lambda-Term M gilt: Y M = M (Y M)
  - Beweisidee: zeige, dass beide Terme auf einen identischen Term reduziert werden können (Übungsaufgabe)
- Der Term Y ist übrigens nicht der einzige Kombinator, der Fixpunkte zu Lambda-Termen konstruiert
  - A. Turing: Θ ≡ (λxy.y(xxy)) (λxy.y(xxy)) (Nachrechnen!)

Rekursion im Lambda-Kalkül

- Die bisher definierten Funktionen waren alle nicht-rekursiv
- Viele Funktionen kann man aber nur unter Zuhilfenahme von Rekursion (bzw. Iteration) beschreiben
- In üblichen Programmiersprachen werden rekursive Funktionsdefinitionen durch die Verwendung von Namen für Funktionen möglich – man verwendet hierbei einfach den Namen der gerade zu definierenden Funktion im Rumpf der Definition:
  - `fun fak(n) -> if (i=0) then 1 else i * fak(i-1).`
- Im Lambda-Kalkül gibt es jedoch keine Namen für Funktionen:
  - Daher stellt man eine rekursive Funktion f mittels einer Funktion G dar, die einen zusätzlichen Parameter g hat, an den man dann G selber bildet
  - Looks complicated, is complicated… ;o)
  - Warum so kompliziert? Damit die Definition von G im eigenen Rumpf verfügbar ist

Rekursive Funktionen sind Fixpunkte

Rekursive Definition von g:

$g = \lambda n. \ldots g \ldots n \ldots$ Rumpf verwendet g

Daraus gewinnt man das Funktional

$G = \lambda g. \lambda n. \ldots g \ldots n \ldots$

Falls G einen Fixpunkt g* hat, d. h. G(g*) = g*, so

$g^* = G(g^*) = \lambda n. \ldots g^* \ldots n \ldots$

Vergleiche: $g = \lambda n. \ldots g \ldots n \ldots$

Rekursive Definition ⇔ Fixpunkt des Funktionals

Beispiel: Fakultät

g = λ n. **if** isZero n **then** c_1 **else** (times n g (pred n)) - rekursiv

G = λ g. λ n. **if** isZero n **then** c_1 **else** (times n g (pred n)) - funktional

Der Fixpunktkombinator dient als Rekursionsoperator

Wir berechnen den gesuchten Fixpunkt des Funktionals G mit dem Fixpunktkombinator, der somit als Rekursionsoperator dient:

Rekursionsoperator

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

$$Y\ f = (\lambda f.\ (\lambda x.\ f\ (\ x\ x))\ (\lambda x.\ f\ (x\ x)))\ f$$

$$\Rightarrow\ (\lambda x.\ f\ (\ x\ x))\ (\lambda x.\ f\ (x\ x))$$

$$\Rightarrow\ f\ ((\lambda x.\ f\ (x\ x\ ))\ (\lambda x.\ f\ (\ x\ x)) \quad \Leftarrow \quad f\ (Y\ f)$$

$$\overset{\beta}{=}$$

also $\quad f\ (Y\ f) = Y\ f$

d.h. $\quad$ **Y f ist Fixpunkt von f**

Beispiel: Fakultät im λ-Kalkül

g = λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))

G = λ g. λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))

} // Keine Lambda-Terme

G = λ g. λ n. (λ a. a) (isZero n) c_1 (times n (g (sub n c_1)))

Y = λf. (λx. f (x x)) (λx.f (x x))

Fak = Y G

Fak c_2 = Y G c_2 $\Rightarrow$ ((λx. G (x x)) (λx. G (x x))) c_2

$\Rightarrow$ G ((λx. G (x x)) (λx. G (x x))) c_2

$\overset{2}{\Rightarrow}$ (isZero c_2) c_1 (times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2)))

$\overset{*}{\Rightarrow}$ times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2))

$\overset{*}{\Rightarrow}$ times c_2 ($\overbrace{(\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))}^{YG\ \Rightarrow}$) c_1

$\overset{*}{\Rightarrow}$ times c_2 ((isZero c_1) c_1 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1))))

$\overset{*}{\Rightarrow}$ times c_2 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1)))

$\overset{*}{\Rightarrow}$ times c_2 (times c_1 ((is_zero c_0) c_1 ...)) $\overset{*}{\Rightarrow}$ c_2

Ausdrucksstärke des Lambda-Kalküls (1)

- Im Folgenden wollen wir zeigen, dass der Lambda-Kalkül genau die rekursiven Funktionen beschreibt (wir übergehen einige Details, die in Vorlesungen zur theoretischen Informatik ggf. nachgeholt werden)
- Eine numerische Funktion ist eine Abbildung f: $\mathbb{N}^k \rightarrow \mathbb{N}$ mit $k \in \mathbb{N} \cup \{0\}$
- Wir definieren hierzu:
  - Anfangsfunktionen:
    - Projektion: $U_i^k(n_1, n_2, \ldots, n_k) = n_i$ für $1 \leq i \leq k$
    - Nullfunktion: $Z(n) = 0$
    - Nachfolger: $S(n) = n + 1$
  - Minimalisierung:
    - Für eine Relation P(m) bezeichne μm[P(m)] die kleinste Zahl m so dass P(m) gilt.
- Bemerkung: im Folgenden notieren wir $n_1, n_2, \ldots, n_k$ kurz mit $\overline{n_k}$
- Eine numerische Funktion ist Lambda-definierbar, wenn es einen Kombinator M gibt, so dass $M\ \overline{n_k} = f(\overline{n_k})$

Ausdrucksstärke des Lambda-Kalküls (2)

- Im folgenden sei C eine Klasse von numerischen Funktionen, und es gelte $g, h, h_1, h_2, \ldots, h_m \in C$
- Wir definieren nun die folgenden Eigenschaften:
  - C ist **abgeschlossen unter Komposition**, wenn für jede Funktion f, die über $f(\overline{n_k}) := g(h_1(\overline{n_k}), \ldots, h_m(\overline{n_k}))$ definiert ist, gilt: $f \in C$
  - C ist **abgeschlossen unter primitiver Rekursion**, wenn für jede Funktion f, die über

$$f(0, \overline{n_k}) = g(\overline{n_k})$$

$$f(j+1, \overline{n_k}) = h(f(j, \overline{n_k}), j, \overline{n_k})$$

    definiert ist, gilt: $f \in C$
  - C ist **abgeschlossen unter unbeschränkter Minimalisierung**, wenn für jede Funktion f, die über $f(\overline{n_k}) = \mu m[\, g(\overline{n_k}, m) = 0]$ definiert ist (wobei für alle $\overline{n_k}$ ein m existiere, so dass $g(\overline{n_k}, m) = 0$ ist), gilt: $f \in C$

Ausdrucksstärke des Lambda-Kalküls (3)

Definition:
Die Klasse der rekursiven Funktionen ist die kleinste Klasse numerischer Funktionen, die alle oben genannten Anfangsfunktionen enthält und abgeschlossen ist unter Komposition, primitiver Rekursion und unbeschränkter Minimalisierung

- Lemma 1: Die Anfangsfunktionen sind Lambda-definierbar
- Beweis:
  - $U_i^k = \lambda x_1\, x_2 \ldots x_k\, .\, x_i$
  - $S = \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$ (siehe succ bei Church-Zahlen)
  - $Z = \lambda\, f\, x\, .\, x$ (siehe c_0 bei Church-Zahlen)

Ausdrucksstärke des Lambda-Kalküls (4)

- Lemma 2: Die Lambda-definierbaren Funktionen sind abgeschlossen unter primitiver Rekursion
- Beweis: Sei f definiert über

$$f(0, \overline{n_k}) = g(\overline{n_k})$$
$$f(j+1, \overline{n_k}) = h(f(j, \overline{n_k}), j, \overline{n_k})$$

und seien g und h Funktionen die (per Induktionsvoraussetzung) durch die Lambda-Terme G und H berechnet werden

  - Intuitiv kann f berechnet werden, indem man überprüft ob j = 0 ist, und wenn ja $g(\overline{n_k})$, ansonsten $h(f(j, \overline{n_k}), j, \overline{n_k})$ berechnet
  - Ein Term M hierfür existiert laut Fixpunktsatz und es gilt:
    $M \equiv Y\ (\lambda\, f\, x\, \overline{y_k}\, .\, If\ (isZero\ x)\ (G\ \overline{y_k})\ (H\ (f(pred\ x)\ \overline{y_k})\ (pred\ x)\ \overline{y_k}\ ))$

Ausdrucksstärke des Lambda-Kalküls (5)

- Lemma 3: Die Lambda-definierbaren Funktionen sind abgeschlossen unter unbeschränkter Minimalisierung
- Beweis:
  - Sei f über $f(n_k) = \mu m[\, g(n_k, m) = 0]$ definiert, wobei g (per Induktionsvoraussetzung) durch den Lambda-Term G berechnet wird
  - Intuitiv kann man f berechnen, indem man bei 0 beginnend für m überprüft, ob $g(n_k, m) = 0$ ist, und wenn ja m ausgibt, ansonsten die Überprüfung mit m + 1 fortsetzt
  - Ein Term für eine solche Funktion kann laut Fixpunktsatz konstruiert werden und man erhält mit Anwendung des Fixpunktkombinators zunächst:
    $N \equiv Y\ (\lambda f\ x_k\ y.\ If\ (isZero\ (G\ x_k\ y))\ y\ (f\ x_k\ (succ\ y)))$
  - Nun definiert man die Funktion f durch den folgenden Term M:
    $M \equiv \lambda\ x_k\ .\ N\ x_k\ c_0$

Ausdrucksstärke des Lambda-Kalküls (6)

$N \equiv Y\ (\lambda f\ x_k\ y.\ If\ (isZero\ (G\ x_k\ y))\ y\ (f\ x_k\ (succ\ y)))$

$M \equiv \lambda\ x_k\ .\ N\ x_k\ c_0$

Nachrechnen:

$$
\begin{aligned}
M\ \overline{n_k} &= N\ \overline{n_k}\ c_0 \\
&= \begin{cases} c_0 & falls\ G\ \overline{n_k}\ c_0 = c_0 \\ N\ \overline{n_k}\ c_1 & sonst \end{cases} \\
&= \begin{cases} c_1 & falls\ G\ \overline{n_k}\ c_1 = c_0 \\ N\ \overline{n_k}\ c_2 & sonst \end{cases} \\
&\ldots
\end{aligned}
$$

- Aus den Lemmata 1 bis 3 folgt nun der Satz:

> Alle rekursiven Funktionen sind Lambda-definierbar.

Noch einmal Auswertungsstrategien (1)

- Bei unserer initialen Betrachtung der Auswertungsstrategien haben wir die volle β-Reduktion und die Normalreihenfolge kennengelernt
- Nun wollen wir unsere Betrachtungen hierzu noch einmal vertiefen und definieren zunächst:
  - Ein Redex wird als **„äußerst“ (outermost)** bezeichnet, wenn er nicht Teil eines anderen Redex ist
  - Ein Redex wird als **„innerst“ (innermost)** bezeichnet, wenn er keinen eigenständigen Redex beinhaltet
- Mit diesen Begriffen können im folgenden die gebräuchlichsten Auswertungsstrategien formuliert werden
  - **Normal Order**: Evaluiere Argumente so oft, wie sie verwendet werden
  - **Applicative Order**: Evaluiere Argumente einmal
  - **Lazy Evaluation**: Evaluiere Argumente höchstens einmal

Noch einmal Auswertungsstrategien (2)

- Eine zentrale Kernfrage: **Welche Auswertungsstrategie führt (möglichst schnell) zu einem nicht mehr weiter reduzierbaren Term?**
  - Bei unserer beispielhaften Berechnung desTerms `Fak` c_2 haben wir nach der initialen Anwendung des Fixpunktkombinators zunächst den Term `isZero` c_2 reduziert
  - Ebenso hätten wir den weiter innen stehenden Fixpunktkombinator zuerst erneut anwenden können (bei voller β-Reduktion kann jeder Term jederzeit reduziert werden)
  - Auf diese Weise hätten wir unendlich oft vorgehen, damit einen immer länger werdenden Term ableiten können und somit nicht das gewünschte Resultat c_2 berechnet
- Eine weitere Kernfrage: Angenommen mehrere unterschiedliche Reduktionsreihenfolgen führen zu einem nicht weiter zu reduzierenden Ergebnis – **führen alle diese Reihenfolgen zum gleichen Ergebnis?**

Noch einmal Auswertungsstrategien (3)

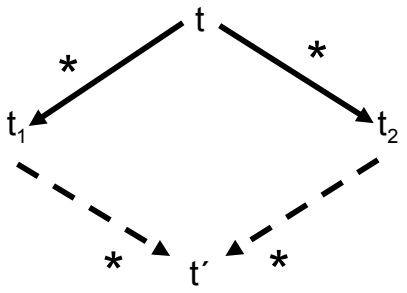
- Wir definieren zunächst einen zentralen Begriff in diesem Zusammenhang:

> Ein Transitionssystem (D, $\rightarrow^*$) heißt genau dann **konfluent**, wenn für alle t, t_1,t_2 ϵ in D gilt: wenn $t \rightarrow^* t_1$ und $t \rightarrow^* t_2$, dann gibt es ein t' ϵ D mit $t_1 \rightarrow^* t'$ und $t_2 \rightarrow^* t'$

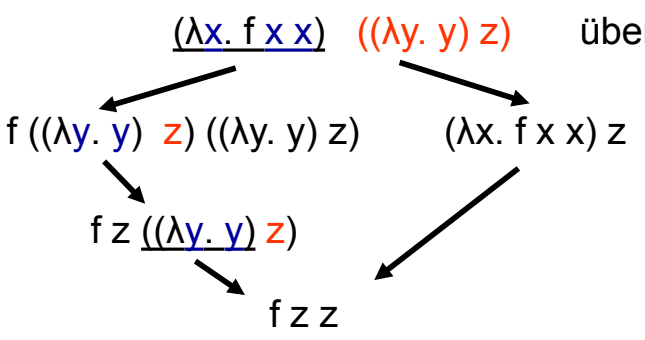
- Wenn der Lambda-Kalkül konfluent ist, kann hieraus gefolgert werden, dass unterschiedliche Reduktionsreihenfolgen, die zu einer nicht mehr weiter zu reduzierenden Form führen, somit auf den gleichen Term führen müssen
- Achtung: hieraus kann nicht gefolgert werden, dass alle Reduktionsreihenfolgen auf den gleichen Term führen, da dies ja nur für „terminierende“ Reduktionsreihenfolgen gilt!

Church-Rosser-Eigenschaft (1)

Satz (Church-Rosser)
Der untypisierte λ-Kalkül ist konfluent:
Wenn $t \overset{*}{\Rightarrow} t_1$ und $t \overset{*}{\Rightarrow} t_2$,
Dann gibt es ein t´ mit $t_1 \overset{*}{\Rightarrow} t´$ und $t_2 \overset{*}{\Rightarrow} t´$.

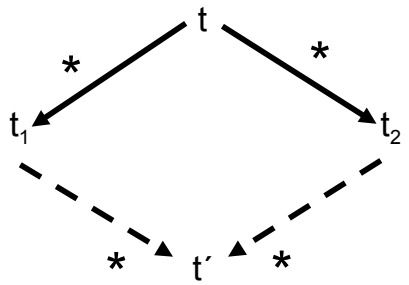


Beispiel

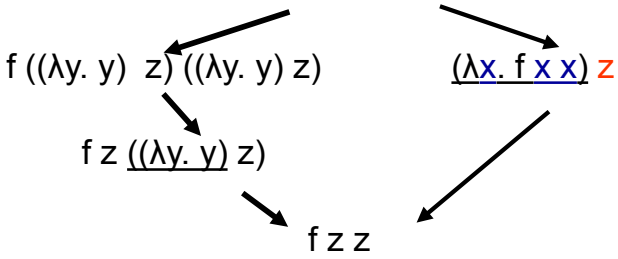


Church-Rosser-Eigenschaft (2)

Satz (Church-Rosser)
Der untypisierte λ-Kalkül ist konfluent:
Wenn $t \overset{*}{\Rightarrow} t_1$ und $t \overset{*}{\Rightarrow} t_2$,
Dann gibt es ein t´ mit $t_1 \overset{*}{\Rightarrow} t'$ und $t_2 \overset{*}{\Rightarrow} t'$.

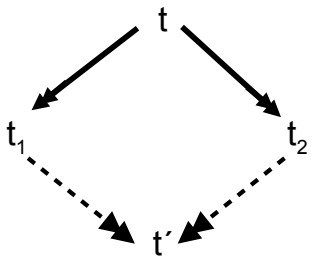


Beispiel (λx. f x x) ((λy. y) z) überlappende Redexe



Beweisidee: Definiere $\twoheadrightarrow$ als „parallele“ β-Reduktion.

- Es gilt: $\Rightarrow \subseteq \twoheadrightarrow \subseteq \overset{*}{\Rightarrow}$
- Zeige „Diamant-Eigenschaft“ für $\twoheadrightarrow$.



Eindeutigkeit der Normalform

Korollar (Eindeutigkeit der Normalform)
Die Normalform eines λ-Terms t ist – sofern sie existiert – eindeutig.

Beweis:

- t_1 und t_2 Normalformen von t, d. h. $t \overset{*}{\Rightarrow} t_1 \not\Rightarrow$ und $t \overset{*}{\Rightarrow} t_2 \not\Rightarrow$
- Nach Church-Rosser gibt es t´mit $t_1 \overset{*}{\Rightarrow} t'$ und $t_2 \overset{*}{\Rightarrow} t'$
- Nach Annahme $t_1 \not\Rightarrow$ und $t_2 \not\Rightarrow$, also $t_1 = t' = t_2$

Bei terminierenden β-Reduktionen ist irrelevant, welchen Redex man zuerst reduziert!

Behandlung von Parametern in Programmiersprachen (1)

- Die Art und Weise, wie in einer Programmiersprache Parameter übergeben – d.h. wie die Reihenfolge und die Zeitpunkte ihrer Auswertung gehandhabt – werden, hat Einfluss auf wichtige Eigenschaften der Sprache:
  - Effizienz der Berechnungen
  - Terminierungsverhalten
  - Ausdruckskraft
- Hierbei ist es insbesondere von Interesse, wie Parameter gehandhabt werden, deren Werte undefiniert sind (z.B. 1/0)

Wir definieren zunächst den zentralen Begriff „strikt“:

Eine n-stellige Funktion heißt **strikt** im k-ten Argument ($1 \leq k \leq n$), wenn gilt: $f(x_1, x_2, \ldots, x_{k-1}, \perp, x_{k+1}, \ldots x_n) = \perp$

- Ein undefiniertes Argument führt hier zu einem undefinierten Resultat

Behandlung von Parametern in Programmiersprachen (2)

- Grundsätzlich kann man die Auswertungsstrategien von Programmiersprachen in **strikte** und **nicht-strikte Strategien** einteilen; sehr gebräuchlich sind dabei insbesondere:
  - **Call by Value:** Ausdrücke, die Parameter bei einem Funktionsaufruf beschreiben, werden vor der Übergabe an die Funktion vollständig ausgewertet
  - **Call by Name:** Ausdrücke, die Parameter bei einem Funktionsaufruf beschreiben, werden nicht bei der Übergabe, sondern erst dann ausgewertet, wenn sie in der aufgerufenen Funktion tatsächlich benötigt werden
- Beide Varianten haben spezifische Vor- und Nachteile:
  - Call by Value: weniger Berechnungsaufwand, wenn ein Parameter mehr als einmal im Funktionsrumpf vorkommt; weniger (Speicher-) Aufwand bei der Übergabe
  - Call by Name: weniger Berechnungsaufwand, wenn ein Argument nichts zum Ergebnis beträgt; höherer Aufwand bei Übergabe

Behandlung von Parametern in Programmiersprachen (3)

- Die Programmiersprache Erlang realisiert grundsätzlich eine strikte Handhabung von Parametern, da sie die Strategie Call by Value verwendet
- Allerdings wird bei der Definition einer Funktion der resultierende Wert erst dann berechnet, wenn die Funktion ausgewertet wird
  - Das erlaubt über den Umweg zusätzlicher Funktionsdefinitionen auch die Realisierung einer nicht-strikten Auswertungsstrategie – ermöglicht Nachbildung der sogenannten **Lazy-Evaluation**
  - Hierbei wird ein nicht-strikt zu evaluierendes Argument als Resultat einer anonymen nullstelligen Funktion (ohne Parameter) „verpackt“
  - Im Rumpf der eigentlichen Funktion wird diese Funktion dann ausgewertet (= aufgerufen), wenn feststeht, dass dieses Argument für die Berechnung des Ergebnises benötigt wird
  - Andere funktionale Sprachen wie Haskell oder Gofer verwenden Call by Name und realisieren damit grundsätzlich Lazy-Evaluation

Behandlung von Parametern in Programmiersprachen (4)

```
-module(lazy).
-export([test1/3, test2/3]).
test1(P, A, B) ->    % A and B are arbitrary values
  if
    P==true  -> A;
    P==false -> B
 end.
test2(P, A, B) ->    % A and B have to be functions
  if
    P==true  -> A();
    P==false -> B()
 end.
```

Behandlung von Parametern in Programmiersprachen (5)

-
```
> lazy:test1(true, 3, 4/0).
** exception error: bad argument in an
   arithmetic expression
   in operator  '/'/2
      called as 4 / 0
```
-
```
> lazy:test2(true, fun() -> 3 end, fun() -> 4/0 end).
3
```
- Erläuterungen:
  - Im zweiten Beispiel wird der Rückgabewert der übergebenen Funktionen nur ausgewertet, wenn sie im Rumpf der auszuführenden Funktion aufgerufen werden
  - Innerhalb von Erlang-Modulen kann man mit Hilfe einer Macro-Definition Schreibarbeit sparen:
```
-define(DELAY(E),fun()->E end).
check() -> test2(true, ?DELAY(3), ?DELAY(4/0)).
```

Behandlung von Parametern in Programmiersprachen (6)

- Je nachdem, ob und wie häufig ein übergebener Parameter im Funktionsrumpf benötigt wird, können bei Lazy-Evaluation Berechnungen
  - komplett eingespart oder
  - (in identischer Form!) wiederholt erforderlich werden
  - Unter Umständen kann man in der betreffenden Funktion durch Einführung einer temporären Variable redundante Mehrfachberechnungen einsparen (→ **Call by Need**)
- Die Parameterübergabe ist bei Call by Name in der Regel aufwendiger als bei Call by Value
  - Die meisten Programmiersprachen (Java, C, C++, Pascal etc.) verwenden daher Call by Value (→ strikte Auswertung)
  - Eine Ausnahme wird oft bei dem If-Konstrukt gemacht (der auszuführende Code ist hier ja meist auch kein Parameter)

Behandlung von Parametern in Programmiersprachen (7)

- Zu Ausdrucksstärke: während strikte Funktionen durch die Strategie Call by Value realisiert werden, ist es nicht so, dass Lazy Evaluation es erlaubt, alle nicht-strikten Funktionen zu realisieren:
  - Die folgenden Gleichungen definieren eine nicht-strikte Multiplikation ⊗ auf der Basis der Multiplikation · für Zahlen:

    $0 \otimes y = 0$

    $x \otimes 0 = 0$

    $x \otimes y = x \cdot y$
  - Wenn ein Argument undefiniert ist, dann liefert ⊗ ein Ergebnis, sofern das andere Argument zu 0 evaluiert wird (→ fak(-1) ⊗ fak(3))
  - Implementiert werden kann die Funktion nur durch eine Art von paralleler Auswertung mit Abbruch der anderen Berechnung sobald 0 als Resultat berechnet und zurückgegeben wurde
- Wir betrachten nun die Beziehungen zwischen Parameter-Behandlung in Programmiersprachen und Reduktion von Lambda-Termen

Auswertungstrategien & Programmiersprachen

Werte in Programmiersprachen wie Haskell:

- Primitive Werte: 2, True
- Funktionen: (\ x → x), (&&), (\x → (\ y → y+y) x)

Werte im λ-Kalkül:

- Abstraktionen: $c_2 = \lambda s.\ \lambda z.\ s\ (\ s\ z)$, $C_{true} = \lambda t.\ \lambda f.\ t$

  $\lambda x.\ x$, $\lambda b_1.\ \lambda b_2.\ b_1\ b_2\ (\lambda t.\ \lambda f.\ f)$, $\lambda x.\ (\lambda y.\ \textit{plus}\ yy)\ x$

Auswertungsstrategie: Keine weitere Reduzierung von Werten

Reduziere keine Redexe unter Abstraktionen (umgeben von λ):

⇒ call-by-name, call-by-value

Call-By-Name

Call-by-name Reduziere linkesten äußersten Redex

- Aber nicht falls von einem λ umgeben

(λy. (λx. y (λz. z) x)) ((λx. x) (λy. y))

⇒ (λx. ((λx. x) (λy. y)) (λz. z) x) ⇏

Intuition: Reduziere Argumente erst, wenn benötigt

Auswertung in Haskell: **Lazy-Evalution = call-by-name (+ sharing)**

- Standard-Auswertungsstrategie für Funktionen / Konstruktoren

```
listOf x = x : listOf x
3 : listOf 3 ⇏
(div 1 0) : (6 : [])
tail ((div 1 0) : (6 : [])) ⇒  6 : [] ⇏
```

Call-By-Value

Call-by-value Reduziere linkesten Redex

- der nicht einen λ umgeben
- und dessen Argument ein *Wert* ist

(λy. (λx. y (λz. z) x)) ((λx. x) (λy. y))
⇒ (λy. (λx. y (λz. z) x)) (λy. y)
⇒ (λx. (λy. y (λz. z) x)) ⇏

Intuition: Argumente vor Funktionsaufruf auswerten
Auswertungsstrategie vieler Sprachen: Java, C, Scheme, ML, ...

Arithmetik in Haskell: Auswertung **by-value**

```
prodOf x = y * prodOf x
3 * prodOf 3 ⇒  3 * (3 * prodOf 3) ⇒ 3 * (3 * prodOf 3)) ⇒ ...
((div 1 0) * 6) * 0 ⇒ ⊥
((div 2 2) * 6) * 0 ⇒ ((1 * 6) * 0 ⇒ 6 * 0 ⇒ 0 ⇏
```

Vergleich der Auswertungsstrategien

Call-by-name vs. Call-by-value:

- Werten nicht immer zur Normalform aus: λx. (λy. y) x
- Gibt es Normalform, dann darauf β-reduzierbar (Church-Rosser)
- Call-by-name terminiert öfter

Y (λy. z) = λf. (λx. f (x x)) (λx. f (x x)) (λy. z)

⇒ λx. (λy. z (x x)) (λx. (λy. z)) (x x)

$\overset{cbn}{\Rightarrow}$ (λy. z) ((λx. (λy. z) (x x)) (λx. (λy. z (x x))) $\overset{cbn}{\Rightarrow}$ z

$\overset{cbv}{\Rightarrow}$ (λy. z) ((λx. (λy. z) (x x)) (λx. (λy. z (x x)))

$\overset{cbv}{\Rightarrow}$ (λy. z) ((λy. z) ((λx. (λy. z) (x x)) (λx. (λy. z (x x))))

...

Standardisierungsatz

Wenn t eine Normalform hat, dann findet Normalreihenfolgenauswertung diese.

Abschließende Bemerkungen

- Der Lambda-Kalkül wurde in den dreißiger Jahren des 20. Jahrhunderts von Alonzo Church erfunden, um damit grundsätzliche Betrachtungen über berechenbare Funktionen anzustellen
- Trotz der Einfachheit der dem Kalkül zugrunde liegenden Regeln, realisiert er ein universelles Berechnungsmodell
- Der Lambda-Kalkül hat die Entwicklung zahlreicher, für die Informatik wichtiger Konzepte beeinflusst:
  - Funktionale Programmiersprachen (die minimalen Funktionen von LISP wurden auf Grundlage des Lambda-Kalküls definiert)
  - Forschung zu Typsystemen für Programmiersprachen
  - Repräsentation von Logik-Termen im Lambda-Kalkül führte zu Theorembeweisern für Logiken höherer Stufen
- Manche „Puristen“ vertreten gelegentlich die Ansicht, dass funktionale Programmiersprachen nicht viel mehr sind, als „Lambda-Kalkül mit etwas syntaktischem Zucker“ :o)

Referenzen

[Erw99] M. Erwig. *Grundlagen funktionaler Programmierung.* Oldenbourg Verlag, 1999.

[Han94] C. Hankin. *Lambda Calculi – A Guide for Computer Scientists.* Clarendon Press, Oxford, 1994.

[Sne10] G. Snelting. *Programmierparadigmen Kapitel 11 – Der untypisierte Lambda-Kalkül.* Vorlesungsfolien, Karlsruher Institut für Technologie, 2010.