0.1 Lambda-Kalkül

"Everything is a function"
Teil von Programmiersprachen:

- ML, Haskel, Scheme
- PIZZA (Erweiterung von Java)

0.1.1 Motivation: Mathematik

$$f(x) = a \cdot x + b$$

$$f = x \mapsto a \cdot x + b$$

$$f(0) = a \cdot 0 + b$$

$$f0 = (x \mapsto a \cdot x + b)0 = a \cdot 0 + b$$

$$g = y \mapsto f(fy)$$

Der Name "Lambda-Kalkül" stammt von seiner Schreibweise:

0.1.2 λ-Kalkül

$$f = \lambda x(a \cdot x + b)$$

$$(\lambda x(2 \cdot x))3 = 2 \cdot 3$$

SML: fn(x) $\Rightarrow$ a * x + b
Haskel: $\backslash x \rightarrow$ a * x + b

0.1.3 Punkt-Notation

$a_1a_2a_3.a_4a_5...$ Wobei nach dem Punkt implizit geklammert wird.

$$f = \lambda x.a \cdot x + b;$$

$$\Leftrightarrow f = \lambda x.(a \cdot x + b);$$

$$(\lambda x(x))(\lambda x(\lambda f(f(x\ f))))$$

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

$$\lambda x\ x\ \lambda x\lambda f.f.xf$$

Der Punkt öffnet eine Klammer, die sich so weit rechts schließt, wie syntaktisch möglich.

0.1.4 Geschichte

Alonzo Church (20'er Jahre)
Peter Landin (60'er Jahre) (Hat eine ganze Programmiersprache auf den λ-Kalkül zurückgeführt.)
John McCarthy (50'er / 60'er Jahre) (LISP)

0.1.5 Ungetypter λ-Kalkül

(Interne) Syntax:

t :: = x	Variable
| λxt	Abstraktion, Funktionkonstruktion, Funktionsausführung
| t t'	Applikation, Funktionsauswertung, Funktionselimination

0.1 Definition. In einem Term λxt ist die Variable x gebunden.
Eine Variable y ist frei in einem Term t, wenn sie nicht durch eine λ-Abstraktion gebunden ist.

Beispiel.

$\lambda x.a \cdot x + b$	freie Variablen: a, b	(Parameter)
	gebundene Variablen: x	
$\lambda x.y(\lambda z(x\,z))$	freie Variablen: y	
	gebundene Variablen: x, y	
$\lambda x.y(\lambda y'(x\,y'))$	freie Variablen: y	
	gebundene Variablen x, y'	

Gebundene Variablen können umbenannt werden, ohne die Bedeutung des Terms zu ändern. (α-Regel)

$$\left(\int_{-\infty}^{+\infty} e^{-x^2}\,dx\right)^2 = \int_{-\infty}^{+\infty} e^{-x^2}\,dx \int_{-\infty}^{+\infty} e^{-x^2}\,dx = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} e^{-x^2-y^2}\,dx\,dy$$

$\lambda xx = \lambda yy$ (gemäß α-Regel)

$$(\lambda x.a \cdot x + b)0 \to a \cdot 0 + b$$

0.2 Definition. (β-Reduktion)
$(\lambda xt)s \to [s/x]t$
$[s/x]t$ (lies "s für x in t") bedeutet:
Ersetze alle (freien) Vorkommen von x in t durch s.
Hierbei ist (λxt) Redex und $[s/x]t$ Redukt

Andere Notationen für Substitution $t[s/x]$ $t[x := s]$ $[x \mapsto s]t$ $[x \backslash s]t$

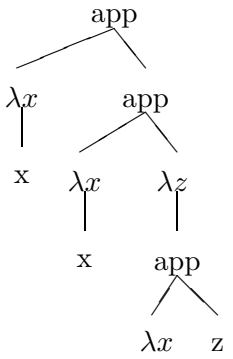
0.3 Definition. (β-Gleichheit)
Zwei Terme t_1, t_2 sind β-gleich $t_1 =_\beta t_2$ falls man t_1 und t_2 β-Reduktionen durchführen kann $t_1 \to_\beta t_1'$ und $t_2 \to_\beta t_2'$, so dass $t_1' = t_2'$.

0.1.6 Operationelle Semantik

Wir unterscheiden 4 Auswertungsstrategien:

Beispiel.
$(\lambda x.x)\ \underline{((\lambda x.x)\lambda z.\underline{(\lambda x x)z})}\quad id = \lambda x.x$

$\underline{id(\underline{id\lambda z.\underline{idz}})}$



A Strategien, die alle Redexe entfernen

1. Volle β-Reduktion
   (Redexe könne in beliebiger Reihenfolge bearbeitet werden.) Nicht deterministisch.
   z.B. $id(id\lambda z.idz) \to id(id\lambda zz) \to id(\lambda zz) \to \lambda z.z$

2. Normale Auswertung / Kopf- (head-)Reduktion/ Leftmost-outermost
   Entfernt den Redex, der am weitesten links-außen steht, zuerst.
   z.B. $id(id\lambda z.idz) \to id(\lambda z.idz) \to \lambda z.idz \to \lambda z.z$

B Strategien, die nicht unter λ reduzieren

3. Call-by-name / lazy / verzögerte Auswertung
   Wie normale Auswertung; stoppt, wenn Wurzel des Termbaums eine Abstraktion.
   z.B. $id(id\lambda z.idz) \to id\lambda z.idz \to \lambda z.idz$

4. Call-by-value / strict / eager / Applikative Reihenfolge
   Zuerst wird Argument ausgewertet, dann in Funktion eingesetzt.
   $id(id\lambda z.idz) \to id\lambda z.idz \to \lambda z.idz$

Beispiel. $hochdrei(x) = x \cdot x \cdot x$
$hochdrei(100!) \overset{cbn}{\to} 100! \cdot 100! \cdot 100!$
$hochdrei(100!) \overset{cbv}{\to} v \cdot v \cdot v (v = 100!)$

$ignore(x) = false$
$ignore(100!) \overset{cbn}{\rightarrow} false$
$ignore(100!) \overset{cbv}{\rightarrow} false(v = 100!)$

0.1.7 Mehrere Argumente

add x y = ...
add = λ x λ y . x + y (Currying)
add (x,y) = ... add = $\lambda z \quad z_1 + z_2$

0.1.8 Programmieren im λ-Kalkül

Der λ-Kalkül ist Turing-Vollständig.

Church-Boolean

if true t_1 $t_2 \rightarrow^+ t_1$
if false t_1 $t_2 \rightarrow^+ t_2$

true = $\lambda t \lambda e.t$
false = $\lambda t \lambda e.e$
if = $\lambda b \lambda t \lambda e.bte$
if true $vw \rightarrow_3$ true $vw \rightarrow_2 v$
if false $vw \rightarrow_3$ false $vw \rightarrow_2 w$
not = $\lambda b \lambda e \lambda t.bte$ not = $\lambda b.ifbfalsetrue \rightarrow_3 \lambda b.b\ false\ true$
and =
or =
Ist jeweils eine schöne Hausaufgabe. ;-(