Lambda-Kalkuel: Cheat Sheet

Allgemein:

α-Äquivalenz

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

η-Äquivalenz

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

β-Reduktion β-Reduktion entspricht der Ausführung der Funktionsanwendung auf einem Redex

Call-by-name Reduziere linkesten äußersten Redex

Call-by-value Reduziere linkesten Redex

Für Haskell gilt: Lazy-Evaluation = call-by-name + sharing
Arithmetik per call-by-value

Rekursionsoperator

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

Church:

Church-Zahlen

Eine (natürliche) Zahl drückt aus, wie oft die Funktion s angewendet wird.

$$c_0 = \lambda s.\, \lambda z.\, z$$
$$c_1 = \lambda s.\, \lambda z.\, s\, z$$
$$c_2 = \lambda s.\, \lambda z.\, s\, (s\, z)$$
$$c_3 = \lambda s.\, \lambda z.\, s\, (s\, (s\, z))$$
$$\vdots$$
$$c_n = \lambda s.\, \lambda z.\, s^n\, z$$

Nachfolgerfunktion:

$$succ = \lambda n.\, \lambda s.\, \lambda z.\, s\, (n\, s\, z)$$

n Church-Zahl,
d.h. von der Form $\lambda s.\, \lambda z.\, \ldots$

Church-Zahl ist Funktion c, welche Funktion s und Startparameter z entgegennimmt und s n mal auf z anwendet, wobei n die von c repraesentierte Zahl ist: churchtoInt c = c (+1) 0

Arithmetische Operationen

Addition:	*plus*	$= \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\, s\, (n\, s\, z)$
Multiplikation:	*times*	$= \lambda m.\, \lambda n.\, \lambda s.\, n\, (m\, s)$
		$\stackrel{\eta}{=} \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, n\, (m\, s)\, z$
Potenzieren:	*exp*	$= \lambda m.\, \lambda n.\, n\, m$
		$\stackrel{\eta}{=} \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, n\, m\, s\, z$

Church-Booleans

`True` wird zu $c_{true} = \lambda t.\, \lambda f.\, t$

`False` wird zu $c_{false} = \lambda t.\, \lambda f.\, f$

Typsysteme:

ohne Typschemata:

Typsystem $\Gamma \vdash t : T$

$\Gamma \vdash t : \tau$ – im Typkontext Γ hat Term t Typ τ.
Γ ordnet freien Variablen x ihren Typ $\Gamma(x)$ zu.

$$\textsc{Const}: \frac{c \in Const}{\Gamma \vdash c : \tau_c} \qquad \textsc{Var}: \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\textsc{Abs}: \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2} \qquad \textsc{App}: \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\, t_2 : \tau}$$

Regeln immer mit neuen Variablen anwenden, nebenbei Constraintmenge aufschreiben, Unifikator ermitteln und auf unterste Variable anwenden

mit Typschemata:

Angepasste Regeln:

$$\textsc{Var}: \frac{\Gamma(x) = \tau' \qquad \tau' \succeq \tau}{\Gamma \vdash x : \tau}$$

$$\textsc{Abs}: \frac{\Gamma, x : \tau_1 \vdash t : \tau_2 \qquad \tau_1 \text{ kein Typschema}}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

Let-Typregel

$$\textsc{Let}: \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \tau_2}$$

Bei let: linke Seite wie gewohnt weitermachen, auf rechter Seite angepasste Regeln verwenden.

Bei Instanziierung "≥" links den allgemeinen Typen (u.U. welcher im linken Teilbaum ermittelt wurde), rechts neue Variablen, Beispiel:

$$\Gamma(k) : \forall \alpha, \beta.\, \alpha \to \beta \to \alpha \geq \beta_4 \to \beta_5 \to \beta_6$$

Contraintmenge und Unifikator wie gewohnt.