Funktionale Programmierung

Sommersemester 2025

Prof. Dr. Jakob Rehof
M.Sc. Felix Laarmann
TU Dortmund
LS XIV Software Engineering

Diese Vorlesung:

- Lambda-Kalkül:
  - Reduktionstheorie (call-by-name, lazy evaluation)
  - Church Numerale

λ - Kalkül: Reduktionstheorie

Syntax (λ-Terme):

$$M ::= x \mid (M\ M) \mid (\lambda x.M)$$

Semantik (Reduktion):

$$((\lambda x.M)\ N) \longrightarrow_\beta M[x := N] \qquad (Redex)$$

λ - Kalkül: Reduktionstheorie

Vollständige Definition der β-Reduktion ($\longrightarrow_\beta$):

- $((\lambda x.P)Q) \longrightarrow_\beta P[x := Q]$
- $P \longrightarrow_\beta P' \Rightarrow (PQ) \longrightarrow_\beta (P'Q)$
- $Q \longrightarrow_\beta Q' \Rightarrow (PQ) \longrightarrow_\beta (PQ')$
- $P \longrightarrow_\beta P' \Rightarrow (\lambda x.P) \longrightarrow_\beta (\lambda x.P')$

Substitution

1.1.13. Definition. For $M, N \in \Lambda^-$ and $x \in V$, the *substitution of* N *for* x *in* M, written $M[x := N] \in \Lambda^-$, is defined as follows, where $x \neq y$:

$$
\begin{array}{lll}
x[x := N] & = N; & \\
y[x := N] & = y; & \\
(P\,Q)[x := N] & = P[x := N]\; Q[x := N]; & \\
(\lambda x.P)[x := N] & = \lambda x.P; & \\
(\lambda y.P)[x := N] & = \lambda y.P[x := N], & \text{if } y \notin \mathrm{FV}(N) \text{ or } x \notin \mathrm{FV}(P); \\
(\lambda y.P)[x := N] & = \lambda z.P[y := z][x := N], & \text{if } y \in \mathrm{FV}(N) \text{ and } x \in \mathrm{FV}(P).
\end{array}
$$

where z is chosen as the $v_i \in V$ with minimal i such that $v_i \notin \mathrm{FV}(P) \cup \mathrm{FV}(N)$ in the last clause.

Alpha-equivalence (alpha-conversion)

1.1.15. Definition. Let α-*equivalence*, written $=_\alpha$, be the smallest relation on Λ^-, such that

$$\begin{array}{ll} P =_\alpha P & \text{for all } P; \\ \lambda x.P =_\alpha \lambda y.P[x := y] & \text{if } y \notin \mathrm{FV}(P), \end{array}$$

and closed under the rules:

$$\begin{array}{lcll} P =_\alpha P' & \Rightarrow & \forall x \in V: & \lambda x.P =_\alpha \lambda x.P'; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & P\ Z =_\alpha P'\ Z; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & Z\ P =_\alpha Z\ P'; \\ P =_\alpha P' & \Rightarrow & P' =_\alpha P; & \\ P =_\alpha P' \ \&\ P' =_\alpha P'' & \Rightarrow & P =_\alpha P''. & \end{array}$$

Terms (modulo alpha-equivalence)

1.1.17. Definition. Define for any $M \in \Lambda^-$, the *equivalence class* $[M]_\alpha$ by:

$$[M]_\alpha = \{N \in \Lambda^- \mid M =_\alpha N\}$$

Then define the set Λ of λ-*terms* by:

$$\Lambda \quad = \quad \Lambda^- / =_\alpha \quad = \quad \{[M]_\alpha \mid M \in \Lambda^-\}$$

1.1.19. Notation. We write M instead of $[M]_\alpha$ in the remainder. This leads to ambiguity: is M a pre-term or a λ-term? In the remainder of these notes, M should always be construed as $[M]_\alpha \in \Lambda$, *except when explicitly stated otherwise.*

Free variables (modulo alpha)

1.1.20. DEFINITION. For $M \in \Lambda$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P) \backslash \{x\}; \\ \mathrm{FV}(P\,Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

1.1.21. REMARK. According to Notation 1.1.19, what we really mean by this is that we define FV as the map from Λ to subsets of V satisfying the rules:

$$\begin{array}{lcl} \mathrm{FV}([x]_\alpha) & = & \{x\}; \\ \mathrm{FV}([\lambda x.P]_\alpha) & = & \mathrm{FV}([P]_\alpha) \backslash \{x\}; \\ \mathrm{FV}([P\,Q]_\alpha) & = & \mathrm{FV}([P]_\alpha) \cup \mathrm{FV}([Q]_\alpha). \end{array}$$

Substitution (modulo alpha)

1.1.22. DEFINITION. For $M, N \in \Lambda$ and $x \in V$, the *substitution of N for x in M*, written $M\{x := N\}$, is defined as follows:

$$
\begin{array}{lcll}
x[x := N] & = & N; & \\
y[x := N] & = & y, & \text{if } x \neq y; \\
(P\,Q)[x := N] & = & P[x := N]\; Q[x := N]; & \\
(\lambda y.P)[x := N] & = & \lambda y.P[x := N], & \text{if } x \neq y, \text{ where } y \notin \mathrm{FV}(N).
\end{array}
$$

1.1.23. EXAMPLE.

(i) $(\lambda x.x\; y)[x := \lambda z.z] = \lambda x.x\; y;$

(ii) $(\lambda x.x\; y)[y := \lambda z.z] = \lambda x.x\; \lambda z.z.$

λ - Kalkül: Reduktionstheorie

Ein term M ist eine *Normalform*, genau dann wenn M kein Redex beinhält (also, es gibt keinen Term Q mit $M \longrightarrow_\beta Q$).

- $(\lambda x.x)$ ist eine Normalform.
- $(y(\lambda x.x))$ ist eine Normalform.
- $((\lambda x.x)y)$ ist keine Normalform.

Ein Term M ist *(schwach) normalisierend*, wenn M eine Normalform hat, das heisst es existiert eine Normalform N mit

$$M \longrightarrow_\beta^* N$$

Nicht alle Terme sind normalisierend. Sei $\Omega = ((\lambda x.(xx))(\lambda x.(xx)))$. Dann gilt offensichtlich

$$\Omega \longrightarrow_\beta \Omega$$

als einzig mögliche Reduktion.

λ - Kalkül: Reduktionstheorie

Terme können beliebig viele Redices beinhalten. Sei

- $\mathbf{I} = (\lambda x.x)$
- $\mathbf{I}^* = ((\mathbf{II})(\mathbf{II}))$
- $\mathbf{K} = (\lambda x.(\lambda y.x))$
- $\mathbf{R} = ((\mathbf{KI})\Omega)$

Dann haben wir

- $\mathbf{I}^* \longrightarrow_\beta (\mathbf{I}(\mathbf{II}))$ und $\mathbf{I}^* \longrightarrow_\beta ((\mathbf{II})\mathbf{I})$
- $\mathbf{R} \longrightarrow_\beta \mathbf{R}$
- $\mathbf{R} \longrightarrow_\beta ((\lambda y.\mathbf{I})\Omega) \longrightarrow_\beta \mathbf{I}$

λ - Kalkül: Reduktionstheorie

Es stellt sich sofort die Frage:

- Sind alle Reduktionspfade gleich gut?

Einige sind *effizienter* als andere, auch mit demselben Ergebnis:

$$\mathbf{R} \longrightarrow_\beta \mathbf{R} \longrightarrow_\beta \mathbf{R} \longrightarrow_\beta^* \mathbf{I}$$

ist nicht so effizient wie die direkte Reduktion zur Normalform, die wir vorher sahen:

$$\mathbf{R} \longrightarrow_\beta^* \mathbf{I}$$

Wir könnten aber auch Fragen:

- Hat jeder normalisierende Term eine *eindeutige Normalform*?

λ - Kalkül: Reduktionstheorie

Die Antwort auf diese letztere Frage ist:

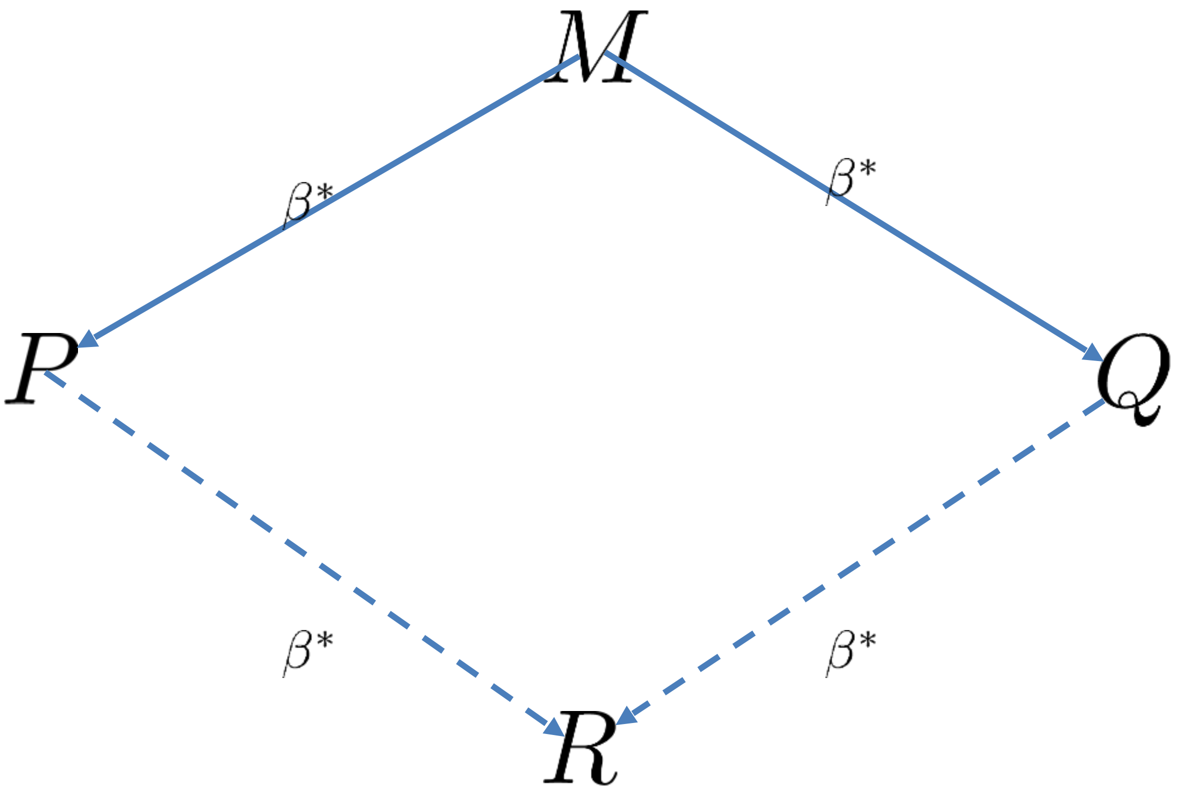
- JA!

Church-Rosser (Konfluenz) Satz:

- *Für alle* M, P, Q *mit* $M \longrightarrow_\beta^* P$ *und* $M \longrightarrow_\beta^* Q$ *existiert* R *mit* $P \longrightarrow_\beta^* R$ *und* $Q \longrightarrow_\beta^* R$

Daraus folgt die Eindeutigkeit von Normalformen unmittelbar.
Somit ist der Kalkül deterministisch bezüglich der Ergebnisse der Berechnung, obwohl die Reduktionssemantik von $\longrightarrow_\beta^*$ nicht deterministisch ist.

λ - Kalkül: Reduktionstheorie



"Church-Rosser-Diamant"

λ - Kalkül: Reduktionstheorie

Wir können nun fragen:

- Gibt es eine deterministische *Reduktionsstrategie*, die bei normalisierenden Termen immer zur Normalform führt?

Die Antwort auf diese Frage ist:

- JA!

Standardisierungssatz:

- *Leftmost-outermost (LO) Reduktion führt immer zur Normalform, wenn sie existsiert.*

λ - Kalkül: Reduktionstheorie

Bei Programmiersprachen betrachtet man nur *weak reduction*, wobei Reduktionen nicht "unter Lambda's geht":

- $((\lambda x.P)Q) \longrightarrow_\beta P[x := Q]$
- $P \longrightarrow_\beta P' \Rightarrow (PQ) \longrightarrow_\beta (P'Q)$
- $Q \longrightarrow Q' \Rightarrow (PQ) \longrightarrow_\beta (PQ')$

λ - Kalkül: Reduktionstheorie

Call-by-name Reduktion ($\longrightarrow_{\beta N}$):

- $((\lambda x.P)Q) \longrightarrow_{\beta N} P[x := Q]$
- $P \longrightarrow_{\beta N} P' \Rightarrow (PQ) \longrightarrow_{\beta N} (P'Q)$

λ - Kalkül: Reduktionstheorie

Call-by-name Reduktion ($\longrightarrow_{\beta N}$):

- $((\lambda x.P)Q) \longrightarrow_{\beta N} P[x := Q]$
- $P \longrightarrow_{\beta N} P' \Rightarrow (PQ) \longrightarrow_{\beta N} (P'Q)$
- Die Relation $\longrightarrow_{\beta N}$ ist die Strategie für sogenannte leftmost-outermost weak head reduction. An expression is said to be in weak head normal form, if it has been evaluated to the outermost data constructor or lambda abstraction (the head).
- Sie wird auch *lazy evaluation* genannt.
- Haskell ist grundsätzlich *lazy*.

λ - Kalkül: Reduktionstheorie

Call-by-value Reduktion ($\longrightarrow_{\beta V}$):

- $((\lambda x.P)V) \longrightarrow_{\beta V} P[x := V]$
- $P \longrightarrow_{\beta V} P' \Rightarrow (PQ) \longrightarrow_{\beta V} (P'Q)$
- $Q \longrightarrow_{\beta V} Q' \Rightarrow (VQ) \longrightarrow_{\beta V} (VQ')$

wobei V (*values*, Werte) durch

- $V ::= x \mid (\lambda x.P)$

definiert sind.
Die Reduktion ist außerdem left-to-right orientiert (dritte Regel).

λ - Kalkül: Datentypen

Das ist ja alles schön und gut, aber wie schreibe ich denn jetzt ein Programm, dass z.B. „1+1“ rechnet?

Hierfür müssen wir Datentypen als Lambda-Terme kodieren!

Heute schauen wir uns natürliche Zahlen an.

Nächste Woche schauen wir uns dann an, wie man (die meisten) Haskell-Datentypen im λ-Kalkül kodieren kann.

Church Numerale

2.14. Definition. (i) $F^n(M)$ with $F \in \Lambda$ and $n \in \mathbb{N}$ is defined inductively as follows.

$$\begin{aligned} F^0(M) &\equiv M; \\ F^{n+1}(M) &\equiv F(F^n(M)). \end{aligned}$$

(ii) The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n \equiv \lambda f x. f^n(x).$$

Church Numerale

PROPOSITION (J.B. Rosser). *Define*

$$\mathsf{A}_+ \equiv \lambda xypq.xp(ypq);$$
$$\mathsf{A}_* \equiv \lambda xyz.x(yz);$$

Then one has for all $n, m \in \mathbb{N}$

(i) $\mathsf{A}_+ c_n c_m = c_{n+m}$.

(ii) $\mathsf{A}_* c_n c_m = c_{n*m}$.

Church Numerale

Jetzt können wir „1+1“ rechnen:

$$
\begin{aligned}
& A_+ \; c_1 \; c_1 \\
= \; & (\lambda x \; y \; p \; q \, . \, x \; p \; (y \; p \; q)) \; (\lambda f \; x \, . \, f \; x) \; (\lambda f \; x \, . \, f \; x) \\
\to_\beta \; & (\lambda y \; p \; q \, . \, (\lambda f \; x \, . \, f \; x) \; p \; (y \; p \; q)) \; (\lambda f \; x \, . \, f \; x) \\
\to_\beta \; & \lambda p \; q \, . \, (\lambda f \; x \, . \, f \; x) \; p \; ((\lambda f \; x \, . \, f \; x) \; p \; q) \\
\to_\beta \; & \lambda p \; q \, . \, (\lambda x \, . \, p \; x) \; ((\lambda f \; x \, . \, f \; x) \; p \; q) \\
\to_\beta \; & \lambda p \; q \, . \, p \; ((\lambda f \; x \, . \, f \; x) \; p \; q) \\
\to_\beta \; & \lambda p \; q \, . \, p \; ((\lambda x \, . \, p \; x) \; q) \\
\to_\beta \; & \lambda p \; q \, . \, p \; (p \; q) \\
= \; & c_2
\end{aligned}
$$