# Der $\lambda$-Kalkül

Frank Huch

Sommersemester 2015

In diesem Skript werden die Grundlagen der Funktionalen Programmierung, insbesondere der $\lambda$-Kalkül eingeführt. Der hier präsentierte Stoff stellt einen teil der Vorlesung Funktionale Programmierung im Sommersemester 2015 dar. Weitere, praktische Inhalte werden in weiteren Dokumenten zur Verfügung gestellt.

## Inhaltsverzeichnis

# 1 Theoretische Grundlagen: Der $\lambda$-Kalkül

**Grundlage** funktionaler Sprachen ist der $\lambda$-Kalkül, der 1941 von Church entwickelt wurde. Motivation war, eine Grundlage der Mathematik und mathematischen Logik zu schaffen; in diesem Zusammenhang entstand der Begriff der „Berechenbarkeit“. Es zeigte sich eine Äquivalenz zur Turing-Maschine, aus der die **Church'sche These** hervorging.

Wichtige Aspekte:

- Funktionen als Objekte
- Gebundene und freie Variablen
- Auswertungsstrategien

## 1.1 Syntax des $\lambda$-Kalküls

Var sei eine abzählbare Menge von Variablen, Exp die Menge der Ausdrücke des reinen $\lambda$-Kalküls, die definiert werden durch ($e \in \text{Exp}$):

$$\begin{array}{rlll} e ::= & v & (\text{mit } v \in \text{Var}) & \text{Variable} \\ | & (e\ e') & (\text{mit } e, e' \in \text{Exp}) & \text{Applikation} \\ | & \lambda v.e & (\text{mit } v \in \text{Var}, e \in \text{Exp}) & \text{Abstraktion} \end{array}$$

Wir verwenden folgende **Konvention** zur Klammervermeidung:

- Applikation ist linksassoziativ, d.h. wir schreiben $xyz$ statt $((xy)z)$
- Wirkungsbereich von $\lambda$ reicht so weit wie möglich, d.h. $\lambda x.xy$ steht für $\lambda x.(xy)$ und nicht für $((\lambda x.x)y)$
- Listen von Parametern: $\lambda xy.e$ statt $\lambda x.\lambda y.e$

**Beachte**: Es gibt keine Konstanten (vordefinierte Funktionen) oder If-Then-Else-Strukturen, diese sind später im reinen $\lambda$-Kalkül definierbar. Der reine $\lambda$-Kalkül ist damit minimal, aber universell, wie wir noch sehen werden.

## 1.2 Substitution

Die **Semantik** des reinen $\lambda$-Kalküls erfolgt wie in funktionalen Sprachen üblich: $\lambda x.e$ entspricht den anonymen Funktion `\x -> e`. Somit ergibt sich in der Semantik: $(\lambda x.x)z \rightsquigarrow z$, wobei im Rumpf ($x$) der Funktion die Variable $x$ durch $z$ substituiert wurde. Ganz so einfach läßt sich diese Regel aber nicht realisieren, wie das folgende Beispiel zeigt. Durch die Anwendung der Regel können Namenskonflikte auftreten:

$$\begin{array}{llll} (\lambda f.\lambda x.fx)x & \not\rightsquigarrow & \lambda x.xx & \text{Konflikt} \\ (\lambda f.\lambda x.fx)x & \rightsquigarrow & \lambda y.xy & \text{kein Konflikt} \end{array}$$

Ein zunächst freie Variable kann durch Anwendung der Reduktionsregel in den Bindungsbereich einer anderen $\lambda$-Abstraktion kommen und so fälschlicher Weise gebunden werden.
Um dies formal fassen zu können, ist es zunächst sinnvoll *freie* bzw. *gebundene Variablen* zu definieren:

$$\begin{array}{ll} \text{free}(v) = \{v\} & \text{bound}(v) = \emptyset \\ \text{free}((e\ e')) = \text{free}(e) \cup \text{free}(e') & \text{bound}((e\ e')) = \text{bound}(e) \cup \text{bound}(e') \\ \text{free}(\lambda v.e) = \text{free}(e) \setminus \{v\} & \text{bound}(\lambda v.e) = \text{bound}(e) \cup \{v\} \end{array}$$

Ein Ausdruck $e$ heißt *geschlossen (Kombinator)*, wenn $\text{free}(e) = \emptyset$ ist.

**Wichtig**: Bei der Applikation dürfen wir Variablen nur Erseten, wenn freie Variablen im aktuellen Parameter nicht gebunden werden. Präzisiert wird dies durch die Definition der *Substitution*:

Seien $e, f \in \text{Exp}$, $v \in \text{Var}$. Dann ist die *Substitution* $e[v/f]$ („ersetze $v$ durch $f$ in $e$“) definiert durch:

$$\begin{array}{rll} v[v/f] & = f & \\ x[v/f] & = x & \text{für } x \neq v \\ (e\ e')[v/f] & = (e[v/f]\ e'[v/f]) & \\ \lambda v.e[v/f] & = \lambda v.e & \\ \lambda x.e[v/f] & = \lambda x.(e[v/f]) & \text{für } x \neq v, x \notin \text{free}(f) \\ \lambda x.e[v/f] & = \lambda y.(e[x/y][v/f]) & \text{für } x \neq v, x \in \text{free}(f), \\ & & \quad y \notin \text{free}(e) \cup \text{free}(f) \end{array}$$

Die letzte Regel sorgt insbesondere dafür, dass keine freien Variablen durch ein anderes $\lambda$ gebunden werden. Stattdessen, benennen wir die gebundene Variable um und machen die Substitution dennoch möglich.
Beachte, dass diese Definition nicht wirklich eine Funktion darstellt, vielmehr eine Relation, da wir nicht vorschreiben, in welche Variable, wir eine gebundene Variable bei einem Namenskonflikt umbenennen.
Kommt eine zu substituierende Variable in einem Ausdruck nicht vor, bleibt der Ausdruck aber unverändert: Falls $v \notin \text{free}(e)$, so ist $e[v/f] = e$.

Mit Hilfe der Substitution können wir nun auch das Ausrechnen der Applikation als Reduktionsrelation formalisieren.

## 1.3 Reduktionsregeln

Definiere die *Beta-Reduktion ($\beta$-Reduktion)* durch folgende Relation:

$$\rightarrow_\beta \subseteq \text{Exp} \times \text{Exp} \text{ mit } (\lambda v.e)f \rightarrow_\beta e[v/f]$$

**Beispiel:** Nun gilt

$$(\lambda f.\lambda x.fx)x \rightarrow_\beta \lambda y.xy$$

aber auch

$$(\lambda f.\lambda x.fx)x \to_\beta \lambda z.xz$$

Somit ist $\to_\beta$ nicht *konfluent*[1]*!*

**Aber**: Die Namen der formalen Parameter spielen keine Rolle bezüglich der Bedeutung einer Funktion, d.h. die syntaktisch verschiedenen Terme $\lambda x.x$ und $\lambda y.y$ sind semantisch gleich (die Identitätsfunktion).

Um dennoch ein konfluentes Ersetzungssystem zu definieren, fürgt man die Umbenennung gebundeneer Variablen als zusätzliche Reduktionsrelation ein:

Die *Alpha-Reduktion ($\alpha$-Reduktion)* ist eine Relation

$$\to_\alpha \subseteq \text{Exp} \times \text{Exp} \text{ mit } \lambda x.e \to_\alpha \lambda y.(e[x/y]) \text{ (falls } y \notin \text{free}(e))$$

**Beispiele:**

$$\lambda x.x \to_\alpha \lambda y.y, \quad \lambda y.xy \to_\alpha \lambda z.xz, \quad \lambda y.xy \not\to_\alpha \lambda x.xx$$

Somit gilt $e \leftrightarrow_\alpha^* e'$ („$e$ und $e'$ sind $\alpha$-äquivalent“) genau dann, wenn sich $e$ und $e'$ nur durch Namen der Parameter unterscheiden.
**Konvention im Folgenden:** Betrachte $\alpha$-äquivalente Ausdrücke als gleich, d.h. rechne auf $\alpha$-Äquivalenzklassen statt auf Ausdrücken.

Die $\beta$-Reduktion kann bisher Ausdrücke ausrechnen, allerdings nur außen – deswegen setzen wir die $\beta$-Reduktion auf beliebige Stellen in Ausdrücken fort (analog für $\alpha$-Reduktion):

$$\begin{array}{lll} \text{Falls } e \to_\beta e' & \text{so auch} & ef \to_\beta e'f \\ & \text{und} & fe \to_\beta fe' \\ & \text{und} & \lambda x.e \to_\beta \lambda x.e' \end{array}$$

Die letzte Fortsetzung der $\beta$-Reduktion im Rumpf einer $\lambda$-Abstraktion lässt man, wenn es um Berechnungen geht, häufig weg. Für Konfluenzbetrachtungen und eine Theorie zur Äquivalenz von $\lambda$-Ausdrücken ist sie aber dennoch wichtig.
**Eigenschaften** der $\beta$-Reduktion:

- Die Relation $\to_\beta$ ist konfluent.
- Jeder Ausdruck besitzt höchstens eine Normalform bezüglich $\to_\beta$ (bis auf $\alpha$-Äquivalenzklassen).

Es gibt allerdings auch Ausdrücke ohne Normalform, dies ist **wichtig** für die Äquivalenz zu Turing-Maschinen! Ein BEispiel stellt der folgende Ausdruck dar:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$$

[1] Eine Relation $\to^*$ ist *konfluent*, falls für alle $u, v, w$ mit $u \to^* v$ und $u \to^* w$ auch ein $z$ existiert mit $v \to^* z$ und $w \to^* z$.

Die darin enthaltene *Selbstapplikation* $(\lambda x.xx)$ würde in Programmiersprachen wie Haskell zu einem Typfehler führen.

**Frage**: Wie findet man die Normalform, falls sie existiert?

Ein *$\beta$-Redex* sei ein Teilausdruck der Form $(\lambda x.e)f$. Dann ist eine *Reduktionsstrategie* eine Funktion von der Menge der Ausdrücke in die Menge der Redexe: Sie gibt an, welcher Redex als nächster reduziert wird. **Wichtige Reduktionsstrategien** sind:

- *Outermost-Strategie (LO, nicht strikt, normal-order-reduction)*: Wähle äußersten Redex
- *Innermost-Strategie (LI, strikt, applicative-order-reduction)*: Wähle innersten Redex

$$\begin{array}{ll}(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \to_{\mathrm{LI}} (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \\ (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \to_{\mathrm{LO}} z\end{array}$$

Da eigentlich alle Funktionensabstraktionen einstellig sind, können die Redexe nicht nach links/rechts unterschieden werden. Dennoch verwendet man den Begriff der LI/LO-Strategie.

**Satz:** Ist $e'$ eine Normalform von $e$, d.h. $e \to_\beta^* e' \not\to_\beta e''$, dann existiert eine LO-Ableitung von $e$ nach $e'$.

Damit berechnet LO jede Normalform, LI manche nicht! LO ist somit *berechnungsstärker* als LI.

Ein wichtiger Aspekt (für Optimierungen, Transformation, Verifikation) ist die **Äquivalenz von Ausdrücken**[2]. Intuitiv sind $e$ und $e'$ genau dann äquivalent, wenn $e$ überall für $e'$ eingesetzt werden kann, ohne das Ergebnis zu verändern.
**Beispiele:**

- $\lambda x.x$ ist äquivalent zu $\lambda y.y$
- $\lambda f.\lambda x.((\lambda y.y)f)((\lambda z.z)x)$ ist äquivalent zu $\lambda f.\lambda x.fx$

**Wünschenswert** wäre es, die Äquivalenz durch syntaktische Transformationen nachweisen zu können. $\alpha$- und $\beta$-Äquivalenz sind dafür aber nicht ausreichend, betrachte folgendes Beispiel (in Haskell-Notation): $(+1)$ ist äquivalent zu $\lambda x.(+)\ 1\ x$, denn bei der Anwendung auf ein Argument $z$ gilt:

$$(+1)\ z \triangleq (+)\ 1\ z \text{ und } (\lambda x.(+)\ 1\ x)z \to_\beta (+)\ 1\ z$$

Doch beide Terme sind weder $\alpha$- noch $\beta$-äquivalent. Daher definieren wir noch die *Eta-Reduktion ($\eta$-Reduktion)* als Relation:

$$\to_\eta \subseteq \mathrm{Exp} \times \mathrm{Exp} \text{ mit } \lambda x.ex \to_\eta e \,(\text{falls } x \notin \mathrm{free}(e))$$

Weitere Sichtweisen der $\eta$-Reduktion:

[2]Wobei wir im Folgenden nur Äquivalenz auf terminierenden Ausdrücken betrachten, sonst ist nicht klar, was man unter „Ergebnis“ versteht.

- Die $\eta$-Reduktion ist eine vorweggenommene $\beta$-Reduktion:
  $(\lambda x.ex)f \to_\beta ef$, falls $x \notin \text{free}(e)$ ist.
- Extensionalität: Funktionen sind äquivalent, falls sie gleiche *Funktionsgraphen* (Menge der Argument-Wert-Paare) haben. Beachte: Falls $fx \leftrightarrow^*_\beta gx$[3] gilt, dann ist $f \leftrightarrow^*_{\beta,\eta} g$ (mit $x \in \text{free}(f) \cap \text{free}(g)$), da gilt:

$$f \leftarrow_\eta \lambda x.fx \leftrightarrow^*_\beta \lambda x.gx \to_\eta g$$

Es gibt auch noch eine weitere Reduktion, die *Delta-Reduktion ($\delta$-Reduktion)*, die das Rechnen mit vordefinierten Funktionen ermöglicht, wie zum Beispiel $(+)\ 1\ 2 \to_\delta 3$. Diese vordefinierten Funktionen sind nicht notwendig, da sie im reinen $\lambda$-Kalkül darstellbar sind.

**Zusammenfassung der Reduktionen** im $\lambda$-Kalkül:

- $\alpha$-Reduktion: Umbenennung von Parametern
- $\beta$-Reduktion: Funktionsanwendung
- $\eta$-Reduktion: Elimination redundanter $\lambda$-Abstraktionen
- $\delta$-Reduktion: Rechnen mit vordefinierten Funktionen

Um zu zeigen, dass wir keine fordefinierten Werte zum Rechnen im $\lambda$-Kalkül benötigen, schauen wir uns eine gängige Codierung von Basisdatentypen an.

## 1.4 Datenobjekte im reinen $\lambda$-Kalkül

**Datentypen** sind Objekte mit Operationen darauf, Idee hier: Stelle die Objekte durch geschlossene $\lambda$-Ausdrücke dar und definiere passende Operationen.

Betrachte den Datentyp der **Wahrheitswerte**: Objekte sind `True` und `False`, die wichtigste Operation ist die `If-Then-Else`-Funktion:

$$\texttt{if\_then\_else}(b, e_1, e_2) = \begin{cases} e_1 & \text{, falls } \quad b = \texttt{True} \\ e_2 & \text{, falls } \quad b = \texttt{False} \end{cases}$$

Daher sind Wahrheitswerte nur Projektionsfunktionen:

$$\begin{array}{rll} \texttt{True} \equiv & \lambda x.\lambda y.x & \text{erstes Argument nehmen} \\ \texttt{False} \equiv & \lambda x.\lambda y.y & \text{zweites Argument nehmen} \end{array}$$

Nun lässt sich eine `If-Then-Else`-Funktion definieren durch

$$\texttt{Cond} \equiv \lambda b.\lambda x.\lambda y.bxy$$

[3] Unter $u \leftrightarrow^*_\beta v$ verstehen wir: es gibt ein $w$, so dass $u \to^*_\beta w$ und $v \to^*_\beta w$.

**Beispiel:**

$$\begin{array}{lll} \texttt{Cond True}\ e_1e_2 & \equiv (\lambda bxy.bxy)(\lambda xy.x)e_1e_2 & \to_\beta^3 (\lambda xy.xe_1e_2) \to_\beta^2 e_1 \\ \texttt{Cond False}\ e_1e_2 & \equiv (\lambda bxy.bxy)(\lambda xy.y)e_1e_2 & \to_\beta^3 (\lambda xy.ye_1e_2) \to_\beta^2 e_2 \end{array}$$

Nun kodieren wir die **natürlichen Zahlen**: Betrachte die *Church-Numerals*, die jede Zahl $n \in \mathbb{N}$ als Funktional darstellen, das eine Funktion $f$ genau $n$ mal auf ein Argument anwendet:

$$\begin{aligned} 0 &\equiv \lambda f.\lambda x.x \\ 1 &\equiv \lambda f.\lambda x.fx \\ 2 &\equiv \lambda f.\lambda x.f(fx) \\ 3 &\equiv \lambda f.\lambda x.f(f(fx)) \\ n &\equiv \lambda f.\lambda x.f^n x \end{aligned}$$

Nun definieren wir Operationen:

- Die wichtigste Operation ist die Nachfolgefunktion `succ`:

  $$\texttt{succ} \equiv \lambda n.\lambda f.\lambda x.nf(fx)$$

  **Beispiel:** Nachfolger von Eins:

  $$\begin{aligned} \texttt{succ}\ 1 &\equiv (\lambda n.\lambda f.\lambda x.nf(fx))(\lambda f.\lambda x.fx) \\ &\to_\beta \lambda f.\lambda x.(\lambda f.\lambda x.fx)f(fx) \\ &\to_\beta \lambda f.\lambda x.(\lambda x.fx)(fx) \\ &\to_\beta \lambda f.\lambda x.f(fx) \equiv 2 \end{aligned}$$

- Test auf Null:

  $$\texttt{is\_Null} \equiv \lambda n.n(\lambda x.\texttt{False})\texttt{True}$$

  **Beispiel:** Teste Null auf Null:

  $$\begin{aligned} \texttt{is\_Null}\ 0 &= (\lambda n.n(\lambda x.\texttt{False})\texttt{True})(\lambda f.\lambda x.x) \\ &\to_\beta (\lambda f.\lambda x.x)(\lambda x.\texttt{False})\texttt{True} \\ &\to_\beta (\lambda x.x)\texttt{True} \\ &\to_\beta \texttt{True} \end{aligned}$$

## 1.5 Mächtigkeit des Kalküls

Ist der reine $\lambda$-Kalkül also berechnungsuniversell? Ja, denn auch Rekursion ist darstellbar, beachte den **Fixpunktsatz**:

**Satz:** Zu jedem $F \in \text{Exp}$ gibt es einen Ausdruck $X$ mit $FX \leftrightarrow_\beta^* X$.

**Beweis:** Wähle zum Beispiel $X = YF$ mit *Fixpunktkombinator* $Y$:

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

Es bleibt zu zeigen, dass $YF \leftrightarrow_\beta F(YF)$ gilt, siehe Übung.

Der reine $\lambda$-Kalkül ist minimal, aber berechnungsuniversell – er besitzt aber eher theoretische Relevanz (Barendregt 1984). Dies wollen wir hier nicht formal zeigen. Die vorgestellten Datenrepräsentationen sollten aber verdeutlichen, dass eine universelle Programmierung möglich ist.
Auch wenn Selbstapplikation in Haskell eigentlich auf Grund des Typsystems nicht möglich ist, können wir den Y-Kombinator mit Hilfe einer Recorddefinition programmieren:

```
newtype Fix a = Fix { app :: Fix a -> a }

fix = \ f -> (\ x -> f (app x x)) (Fix (\ x -> f (app x x)))
```

Da die Definition der Subtraktion bzw. des Dekrementierens für Churchzahlen etwas komplizierter ist, verwenden wir den definierten Fixpunktkombinator zur Definition der Fakultätsfunktion auf Haskell-Zahlen:

```
fac = fix (\ f -> \ x -> if x==0 then 1 else x*f (x-1))
```

Wir definieren ein Funktional für die Fakultätsfunktionen, dessen kleinesten Fixpunkt wir mittels des Y-Kombinators an jeder beliebigen Stelle berechnen können.

## 1.6 Angereicherter $\lambda$-Kalkül

Für funktionale Programmiung ist aber eher ein angereicherter $\lambda$-Kalkül relevant, bei dem Konstanten (vordefinierte Objekte und Funktionen), `let`, `if-then-else` und ein Fixpunktkombinator hinzugenommen werden.

Die **Syntax** des angereicherten $\lambda$-Kalküls:

| | | |
|---|---|---|
| $e ::=$ | $v$ | Variable |
| $\mid$ | $k$ | Konstantensymbol |
| $\mid$ | $(e\ e')$ | Applikation |
| $\mid$ | `let` $v = e$ `in` $e'$ | lokale Definitionen |
| $\mid$ | `if` $e$ `then` $e_1$ `else` $e_2$ | Alternative (manchmal auch `case`) |
| $\mid$ | $\mu v.e$ | Fixpunktkombinator |

Als operationale **Semantik** benutzen wir zusätzlich zur $\alpha$-, $\beta$- und $\eta$-Reduktion noch folgende Regeln:

- $\delta$-Reduktion für Konstanten, z.B. $(+)\ 21\ 21 \rightarrow_\delta 42$
- `let` $v = e$ `in` $e' \rightarrow e'[v/e]$

- `if True then` $e_1$ `else` $e_2 \to e_1$
  `if False then` $e_1$ `else` $e_2 \to e_2$
- $\mu v.e \to e[v/\mu v.e]$

**Beispiel:** Fakultätsfunktion:

$$\texttt{fac} \equiv \mu f.\lambda x.\ \texttt{if}\ ((==)\ x\ 0)\ \texttt{then}\ 1\ \texttt{else}\ ((*)\ x\ (f\ ((-)\ x\ 1)))$$

Der angereicherte $\lambda$-Kalkül bildet die Basis der **Implementierung funktionaler Sprachen** (Peyton, Jones 1987):

1. Übersetze Programme in diesen Kalkül (Core-Haskell im `ghc`):
   - Pattern-Matching wird in `if-then-else` (oder `case`) übersetzt,
   - $f\,x_1\,\ldots\,x_n = e$ wird übersetzt zu $f = \lambda x_1 \ldots x_n.e$,
   - für rekursive Funktionen führt man Fixpunktkombinatoren ein,
   - ein Programm entspricht einer Folge von `let`-Deklarationen und einem auszuwertenden Ausdruck.
2. Implementiere den Kalkül durch eine spezielle abstrakte Maschine, z.B. durch die SECD-Maschine von Landin im `ghc` oder durch eine Graphreduktionsmaschine.

**Weitere Anwendungen** des erweiterten $\lambda$-Kalküls sind die denotationelle Semantik und die Typisierung (getypte $\lambda$-Kalküle).