Das gefürchtete λ-Kalkül

Eine Einführung von:
Fabian Nilius

Übersicht

1. Grundlagen des λ-Kalküls
2. Eine genauere Betrachtung der Grundlagen
3. Feinheiten des λ-Kalküls
4. Zusammenfassung und Ausblick, Literatur

1. Grundlagen des λ-Kalküls

Inhalte:

- Formale Modelle
- Imperative und Funktionale Sprachen
- Grundlagen des λ-Kalküls
- Definition: Kalkül

Ausblick

Wir geben eine erste Einführung in das λ-Kalkül, ein Modell zur formalen Beschreibung von Programmiersprachen.

Dies beinhaltet eine kurze Motivation formaler Modelle, eine Übersicht über die Hauptunterschiede zwischen imperativer und funktionaler Programmierung und eine Definition des Begriffes „Kalkül“.

Formale Modelle

- Die Vielfalt von existierenden Programmiersprachen und Rechnerarchitekturen macht die Einführung von **formalen Modellen** sinnvoll,
  - die einerseits **mächtig** sind, um die Programmiersprachen auf sie zurückzuführen,
  - und andererseits **einfach**, um Sätze und Beweise über sie nicht unnötig komplex werden zu lassen.

Turingmaschine und Lambda-Kalkül

- Zwei der bekanntesten Modelle sind
  - die **Turingmaschine**, verwandt mit imperativen Sprachen wie Pascal oder C.
  - der **Lambda-Kalkül**, verwandt mit funktionalen Sprachen, wie Miranda, ML oder Lisp (letztere eingeschränkt).

Imperative und funktionale Sprachen

- In **imperativen** Sprachen
  - werden Programme als eine **Folge** von Anweisungen interpretiert, die nacheinander abgearbeitet werden
  - können Variablen **wechselnde** Werte zugewiesen werden
- In **funktionalen** Sprachen hingegen
  - besteht ein Programm *allein* aus Funktions**definition**, Funktions**anwendung** und Funktions**komposition**
  - steht jeder Bezeichner in einen Kontext für **genau einen** Wert (wie in der Mathematik)

Ein funktionales Programm

Funktionen können Parameter sein!

```
square (n)  = (n * n)
twice (f  x)  = f (f (x))
```

Funktionsdefinition

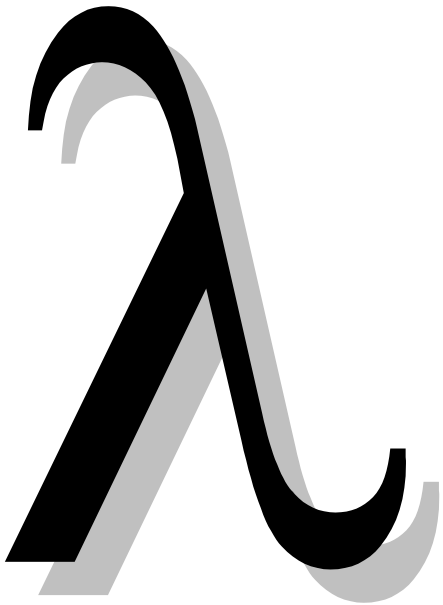
```
square (3)
twice (square 3)
```

Funktionsanwendung $\rightarrow$ 9

Anwendung + Komposition $\rightarrow$ 81

Die Ausführung eines funktionalen Programms erfolgt, indem ein Ausdruck ausgewertet wird und einen Wert liefert.

We now proudly present:



Lambda-Kalkül

Der Lambda-Kalkül besteht aus zwei Bausteinen:

- **Funktionsabstraktion**

  $\lambda x . A$ definiert eine **(anonyme)** Funktion, die ein x bekommt, und einen Ausdruck A als Funktionskörper hat (in dem in der Regel x vorkommt, aber nicht vorkommen muß)

- **Funktionsapplikation**

  $F\ A$ bedeutet, daß die Funktion F auf den Ausdruck A angewandt wird

Beispiele für Funktionen (1)

- Die Identität:

  $\lambda x . x$

- Eine Funktion, die jedes Argument auf die Identitätsfunktion abbildet:

  $\lambda y . (\lambda x . x)$

- Die Identität, angewandt auf sich selbst:

  $(\lambda x . x)\ (\lambda y . y) \quad \rightarrow \quad (\lambda y . y)$

Die Bezeichner der Variablen (x bzw. y) sind hier austauschbar

Beispiele für Funktionen (2)

Nicht alle Variablennamen müssen definiert sein, auch die folgenden sind korrekte λ-Terme:

$\lambda x . f x$

$(\lambda y . y)\ (z)$ $\quad (\rightarrow\ z)$

$u\ v$

$\lambda k . j\ m$

Beispiele für Funktionen (3)

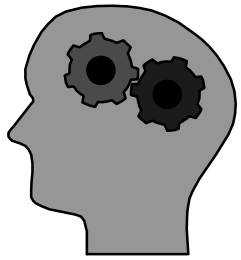
Ein komplexerer Ausdruck:

- $(\lambda f . (\lambda x . f\ (f\ x)))\ u\ v$

Die Variable f ist auch in den inneren Ausdrücken bekannt!

$(\lambda f . (\lambda x . f\ (f\ x)))\ u\ v$

$\rightarrow \quad (\lambda x . u\ (u\ x))\ v$



$\rightarrow \quad u\ (u\ v)$

Diese Funktion wendet also eine Funktion zweimal auf ein Argument an.

Eigenschaften des λ-Kalküls

- Als Bausteine gibt es nur Funktionabstraktion und -applikation. Das ist alles!

  Es gibt **keine** Zahlen, Funktionsnamen, Arithmetische Funktionen, Wahrheitwerte.

- Die Funktionen werden nicht benannt, sie sind **anonym**.
- Der Lambda-Kalkül ist **ungetypt**.
- Die Definition der Funktionen wird mit den Parametern zu **einem** λ–Ausdruck zusammengefaßt. Es gibt keine Trennung zwischen Programmdefinition und einer Anwendung des Programms.

Kalkül

Definition:

Ein **Kalkül** besteht aus zwei wesentlichen Teilen:

- Der **Kalkülsprache**, d.h.
  - einem zugrundeliegenden **Alphabet** von Zeichen
  - einer Definition der **wohlgeformten** Ausdrücke
- Dem **Deduktionsgerüst**, d.h.
  - **Axiomen** aus der Menge der wohlgeformten Ausdrücke (Terme)
  - **Ableitungsregeln,** mit denen Ausdrücke umgeformt werden

nach K.Schröter (1941)

Anmerkung: Diese Seite soll hauptsächlich der Erklärung des Namens Lambda-Kalkül dienen, deshalb verzichten wir auf eine formale Definition der Begriffe Alphabet, Zeichen, Ausdruck etc.

Abstrakte Syntax

Die abstrakte Syntax des λ-Kalküls (Definition aller syntaktisch wohlgeformten Ausdrücke):

< λ-Ausdruck>	=	<Variable>	(Variablenname)
	|	<λ-Ausdruck> < λ-Ausdruck>	(Applikation)
	|	λ < Variable> . <λ -Ausdruck>	(Abstraktion)

Diese *abstrakte Syntax* verzichtet (im Gegensatz zur *konkreten Syntax*) auf die Angabe von Klammern und der konkreten Zeichen in Variablennamen.

Die Axiome

Die Axiome unseres Kalküls sind die Ausdrücke, die wir als gegeben setzen, also die Funktionsdefinitionen und die Funktionsanwendungen.

Sie lassen sich mit den jetzt folgenden Ableitungsregeln umformen, so daß über die Ableitungsregeln z.B. aus einem Ausdruck sein dazugehöriger Wert ableitbar ist.

$(\lambda\, x\, .\, x)\ (\lambda\, y\, .\, y) \quad \rightarrow \quad (\lambda\, y\, .\, y)$

Axiom Ableitung

Die Ableitungsregeln

Wir stellen jetzt die Ableitungs- bzw. Umformungsregeln unseres Kalküls vor, geben aber erst später eine formal exakte Definition.

Die zwei grundlegenden Regeln,

- α-Konversion und
- β-Konversion

ergeben sich, wie wir sehen werden, direkt aus unseren Beispielen.

α-Konversion

Variablennamen sind austauschbar. Dieses ist die erste Umwandlungsregel unseres Kalküls:

$$(\lambda\, x\, .\, A) \quad \leftrightarrow \quad (\lambda\, y\, .\, A')\,,$$

wobei in A' alle Vorkommnisse von x durch y ersetzt wurden.

Dies haben wir bei den beiden Versionen der Identitätsfunktion benutzt:

$$(\lambda\, x\, .\, x) \quad \underset{\alpha}{\leftrightarrow} \quad (\lambda\, y\, .\, y)$$

β-Konversion

Mit β-Konversion ist die Anwendung (Applikation) einer Funktion gemeint. Sie ist die zweite Umwandlungsregel zwischen Ausdrücken, die wir kennenlernen:

$$(\lambda x . A)\ (B) \quad \overset{\beta}{\leftrightarrow} \quad A' ,$$

wobei in A' alle Vorkommnisse von x durch B ersetzt wurden.

Wir haben die β-Konversion schon in Aktion gesehen, als wir die Identitätsfunktion auf sich selber angewendet haben:

$$(\lambda x . x)\ (\lambda y . y) \quad \overset{\beta}{\leftrightarrow} \quad (\lambda y . y)$$

2. Eine genauere Betrachtung der Grundlagen

Inhalte:

- Ersetzung von Variablen durch die Parameter
- Auftauchende Probleme
- Eine Ersetzungsfunktion
- Formale Definition von α- und β-Konversion

Ausblick

Wir haben bisher nicht genau definiert, wie bei einer Funktionsanwendung die Variablen durch die Argumente ersetzt werden sollen.

Dies holen wir jetzt nach.

Dafür betrachten wir erst, welche Probleme auftreten können, und definieren dann eine Ersetzungsfunktion.

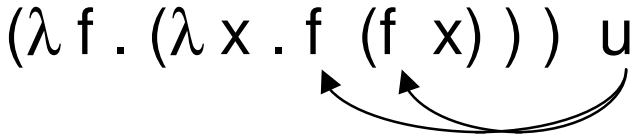
Mit dieser Funktion definieren wir dann auch die α- und β-Konversion formal.

Sichtbarkeit

Schachteln wir Definitionen von Funktionen (d. h., wir definieren eine Funktion im Funktionskörper einer anderen), sind - wie von normalen Programmiersprachen gewohnt - die weiter außen definierten Variablennamen innen sichtbar.

Dies drückt sich in der Regel "β-Konversion" dadurch aus, daß die Argumente für *alle* Vorkommnisse der Variable eingesetzt werden.

Das u wird für alle inneren f eingesetzt:



Gebundene und freieVariablen

Betrachen wir nur den Funktionskörper der Funktion eben:

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

Das f wird weiter außen definiert und erscheint hier wie eine globale Variable, sie ist in diesem Ausdruck **nicht gebunden (frei)**.

x hingegen gehört zu einer Funktionsabstraktion in diesem Ausdruck und ist **gebunden**.

In dem Ausdruck $u\ (\lambda u . u\ v)$ sind das äußere u und das v freie Variablen. Das innere u bezieht sich hingegen auf die innen definierte Funktion, und ist an diese gebunden.

Formale Definition von “frei”

x ist frei in x (aber nicht in einer anderen Variable)

x ist frei in E F $\Leftrightarrow$ x frei in E oder
x frei in F

x ist frei in λ y. E $\Leftrightarrow$ x $\neq$ y und x frei in E

Mehrfach benutzte Namen

Es kann jetzt vorkommen, daß in einer inneren Funktionsdefinition die gleichen Variablennamen benutzt werden, wie in der äußeren.

Ersetzen wir jetzt die Variablen bei einer Funktionsanwendung durch die übergebenen Argumente, so wollen wir nur die Vorkommnisse ersetzen, die sich auf die äußere Funktion beziehen.

Wir ersetzen also nur diejenigen Variablen, die im Funktionskörper **frei** vorkommen.

Beispiel für Ersetzung

$(\lambda f . f\ (\lambda f . f)\ )\ A$

Richtig:

$\rightarrow\quad A\ (\lambda f . f)$

Falsch:

$\rightarrow\quad A\ (\lambda f . A)$

Falsch:

$\rightarrow\quad A\ (\lambda A . A)$

Namensprobleme

Obwohl Variablennamen nach der α-Konversion beliebig austauschbar sind, müssen wir, wie eben gesehen, aufpassen, wenn Namen mehrfach benutzt werden.

Dieses kann noch in einer zweiten Form auftreten:

Das Argument kann freie Variablen enthalten, die sich auf einen äußeren Kontext beziehen.

An der Stelle, an der dieses Argument eingesetzt wird, können lokal in einer Funktion die gleichen Variablennamen mit anderer Bedeutung benutzt werden.

Beispiel

Im folgenden (nicht sinnvollem, aber relativ einfachen) Ausdruck wird die Problematik deutlich:

$\lambda x . (\lambda y . (\lambda x . y)\ x)$

Das äußere, ganz links stehende x ist noch an keinen Parameter gebunden, wir können aber den inneren Ausdruck weiter auswerten:

Hier wird das y an das äußere x gebunden. Setzen wir aber einfach das x für das y ein, so bezieht sich das x plötzlich auf die innerste Funktionsdefinition!

Lösung

Die einzige Lösung ist, daß wir die inneren Variablennamen umbenennen:

$\lambda\, x\, .\, (\lambda\, y\, .\, (\lambda\, x'\, .\, y)\ \ x\ )$

Dann kann, wenn wir in dem Teilausdruck $(\lambda\, x'\, .\, y)$
y durch x ersetzen wollen, nichts mehr schiefgehen.

Ersetzungsfunktion

Zusammenfassend definieren wir jetzt die folgende Ersetzungsfunktion:

Definition: Die Funktion **A[x := B]**, bei der die Variable x in A durch B ersetzt wird, wird definiert durch:

x [x := B]	= B	(meint: falls A = x)
c [x := B]	= c	c ist Variable verschieden von x
E F [x := B]	= E [x := B] F [x := B]	
(λ x . E) [x := B]	= λ x . E	
(λ y . E) [x := B]	= λ y . (E [x := B]),	falls x garnicht in E vorkommt oder y nicht frei in B vorkommt
(λ y . E) [x := B]	= λ z . ((E [y := z]) [x := B]),	anderenfalls wobei z eine neue Variable ist

α- und β-Konversion formal

Mit der Ersetzungsfunktion können wir unsere Konversionen formal definieren:

Austausch von Variablennamen:

α- Konversion: $(\lambda\, x\, .\, E) \underset{\alpha}{\leftrightarrow} (\lambda\, y\, .\, E[x := y])$,
falls y nicht frei in E vorkommt

Funktionsanwendung:

β- Konversion: $(\lambda\, x\, .\, E)\, F \underset{\beta}{\leftrightarrow} E[x := F]$

3. Feinheiten des λ-Kalküls

Inhalte:

- Mehrere Parameter - Currying
- Fallunterscheidungen
- Natürliche Zahlen
- Arithmetische Funktionen
- Rekursion

Ausblick

Bisher haben wir schon einige Funktionen im λ-Kalkül betrachtet, diese waren aber alle sehr primitiv und nicht von praktischem Nutzen.

In diesem Abschnitt zeigen wir, daß sich auch Fallunterscheidungen, Zahlen, arithmetische Funktionen im λ-Kalkül notieren lassen.

Beginnen werden wir mit einer Vorstellung von Funktionen mit mehreren Parametern, und enden wird der Abschnitt mit Rekursion im λ-Kalkül.

Currying - nach H.B. Curry

- Verlassen wir kurz unser Kalkül, und betrachten wir die folgende Situation:

  add 3 4 liefert, wie erwartet: 7

- Was aber liefert add 3 ?

  Wir können uns als Ergebnis eine Funktion vorstellen, die zu jeder beliebigen Zahl 3 addiert, etwa so:

  add 3 $\rightarrow$ addiereDrei

- Dann würde unsere obige Anwendung mit 4 so aussehen:

  add 3 4 $\rightarrow$ addiereDrei 4 $\rightarrow$ 7

Currying (2)

Eine Funktion nach dem Curry-Prinzip nimmt also ein Argument nach dem anderen und liefert jeweils eine neue Funktion, bis alle Argumente verwendet sind.

Mathematisch gesehen:

Zu einer Funktion $f: X \times Y \rightarrow Z$

bezeichnet für ein beliebiges, aber festes x $f_x: Y \rightarrow Z$

eine einstellige Funktion, die einem y einen Wert $f(x\ y)$ zuordnet.

Abstraktion mit mehreren Parametern

Das Curry-Prinzip führt also jede Anwendung einer mehrstelligen Funktion auf eine Folge von Anwendungen einstelliger Funktionen zurück.

Im λ-Kalkül haben wir nur einstellige Funktionen. Wir benutzen das Curry-Prinzip, um auch mehrstellige Funktionen zu erhalten:

$$\lambda\, x_1\, x_2 \ldots x_n \,.\, A \quad := \quad \lambda\, x_1 \,.\, (\lambda\, x_2 \,.\, \ldots \,(\lambda\, x_n \,.\, A)..))$$

Wird x_1 an einen Wert gebunden, so erhalten wir eine Funktion zurück, die einen Parameter weniger erwartet.

Die Funktionsabstraktion ist, wie zu sehen, **rechtsassoziativ**.

Applikation mehrerer Parameter

Hat eine Funktion mehrere Parameter, so führen wir für die Applikation folgende Schreibkonvention ein:

$$F\ x_1\ x_2\ \ldots\ x_n \quad := \quad (..((F\ x_1)\ x_2)\ \ldots\ x_n)$$

Die Funktionsapplikation ist **linkssassoziativ**.

Dies entspricht der Vorstellung $f\ x\ y = (f\ x)\ y = f_x\ y$.

Ein Beispiel für mehrere Parameter

Mit diesen Konventionen können wir eine uns schon bekannte Funktion einfacher schreiben:

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

Rechts sind die übrig gebliebenen Klammern nötig, da wir hier nicht f auf f anwenden wollen, sondern zweimal auf x.

Logische Funktionen - True, False, If

Wir identifizieren jetzt die Werte *True* und *False* mit zwei bestimmten Funktionen:

True sei die folgende Funktion:

λ x y . x (Projektion auf erstes Argument)

False sei die folgende Funktion:

λ x y . y (Projektion auf zweites Argument)

Damit können wir ein **If** definieren:

If .. then .. else .. entspricht hier λ b x y . b x y,

wobei b entweder True oder False ist.

Logische Funktionen - Beispiel

If True then A_1 else A_2

übersetzt in das λ-Kalkül liefert:

	If	True	A_1	A_2
	$(\lambda\ b\ A\ A'.\ b\ A\ A')$	$(\lambda\ x\ y\ .\ x)$	A_1	A_2
$\rightarrow$	$(\lambda\ A\ A'.\ (\lambda\ x\ y\ .\ x)\ A\ A')$		A_1	A_2
$\rightarrow$	$(\lambda\ A'.\ (\lambda\ x\ y\ .\ x)\ A_1\ A')$			A_2
$\rightarrow$	$(\lambda\ x\ y\ .\ x)\ A_1\ A_2)$			
$\rightarrow$	$(\lambda\ y\ .\ A_1)\ A_2$			
$\rightarrow$	A_1			

Natürliche Zahlen

Im λ-Kalkül gibt es nur Funktionen. Um Zahlen zu erhalten, werden wir jetzt ganz bestimmte Funktionen mit den natürlichen Zahlen identifizieren:

$F^n(A)$ (die n-malige Anwendung von F auf A) wird induktiv definiert durch:

$$F^0(A) := A \qquad F^{n+1}(A) := F(F^n(A))$$

Die Zahlen $z_0, z_1, z_3 \ldots$ werden dann definiert durch:

$$z_n \equiv \lambda f\, x .\ f^n(x)$$

Beispiele für Zahlen

Meinen wir die Zahl Drei (z_3), so schreiben wir also:

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

Die Null (z_0) wäre:

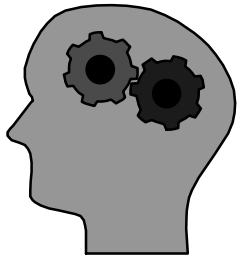
$\lambda f x . x$

Für die Zweiundvierzig (z_{42}) schreiben wir:

$\lambda f x . f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (

Arithmetische Funktionen - Addition

Mit den eben definierten Zahlen wollen wir natürlich auch rechnen. Unsere Addition sieht so aus:



(Für zwei Zahlen z und z',
f und x sind die Parameter der entstehenden Zahl):

Definition von Plus: $\lambda\ z\ z'\ f\ x\ .\ z\ f\ (z'\ f\ x)$

Addition - Beispiel

Definition von Plus: $\lambda\ z\ z'\ f\ x\ .\ z\ f\ (z'\ f\ x)$

Am Beispiel: Plus $z_2\ z_3$ $(\rightarrow\ z_5)$

$z_2 = (\lambda\ f\ x\ .\ f\ (f\ (x)))$

$z_3 = (\lambda\ f\ x\ .\ f\ (f\ (f\ (x))))$

	Plus	z_2	z_3
	$(\lambda\ z\ z'\ f\ x\ .\ z\ f\ (z'\ f\ x))$	z_2	z_3
$\rightarrow$	$(\lambda\ z'\ f\ x\ .\ z_2\ f\ (z'\ f\ x))$		z_3
$\rightarrow$	$(\lambda\ z'\ f\ x\ .\ f\ (\ f\ (z'\ f\ x)))$		z_3
$\rightarrow$	$(\lambda\ f\ x\ .\ f\ (\ f\ (z_3\ f\ x)))$		
$\rightarrow$	$(\lambda\ f\ x\ .\ f\ (\ f\ (\ f\ (\ f\ (\ f\ (x))))))$	$(= z_5)$	

Addition - Beispiel (ausgeschriebene Version)

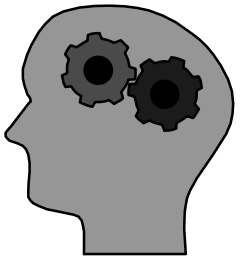
Definition von Plus: $\lambda\, z\ z'\ f\ x\ .\ z\ f\ (z'\ f\ x)$

Am Beispiel: Plus z_2 z_3 $(\rightarrow\ z_5)$

Plus	z_2	z_3
$(\lambda\, z\ z'\ f\ x\ .\ z\ f\ (z'\ f\ x))$	$(\lambda\, g\ y\ .\ g\ (g\ (y)))$	$(\lambda\, g\ y\ .\ g\ (\ g\ (\ g\ (y))))$
$\rightarrow\ (\lambda\, z'\ f\ x\ .\ (\lambda\, g\ y\ .\ g\ (g\ (y)))\ f\ (z'\ f\ x))$		$(\lambda\, g\ y\ .\ g\ (\ g\ (\ g\ (y))))$
$\rightarrow\ (\lambda\, z'\ f\ x\ .\ (\lambda\, y\ .\ f\ (\ f\ (y))\ (z'\ f\ x))$		$(\lambda\, g\ y\ .\ g\ (\ g\ (\ g\ (y))))$
$\rightarrow\ (\lambda\, z'\ f\ x\ .\ f\ (\ f\ (z'\ f\ x)))$		$(\lambda\, g\ y\ .\ g\ (\ g\ (\ g\ (y))))$
$\rightarrow\ (\lambda\, z'\ f\ x\ .\ f\ (\ f\ (\lambda\, g\ y\ .\ g\ (\ g\ (\ g\ (y)))\ f\ x)))$		
$\rightarrow\ (\lambda\, f\ x\ .\ f\ \ (\ f\ (\lambda\, y\ .\ f\ (\ f\ (\ f\ (y)))\ x)))$		
$\rightarrow\ (\lambda\, f\ x\ .\ f\ (\ f\ (\ f\ (\ f\ (\ f\ (x))))))$	$(=z_5)$	

Arithmetische Funktionen - Mal

Weiter definieren wir die Multiplikation:



(Für zwei Zahlen z und z',
f und x sind die Parameter der entstehenden Zahl):

Definition von Mal: $\lambda\ z\ z'\ f\ x\ .\ z\ (z'\ f)\ x$

Dies läßt sich auch schreiben als: $\lambda\ z\ z'\ f\ .\ z\ (z'\ f)$,
da dieser Ausdruck eine 1-stellige Funktion zurückgibt.

Eine Beispielrechnung für die Multiplikation sei als
Übungsaufgabe dem geneigten Leser überlassen. :-)

Benannte Funktionen

Unsere Funktionen haben keinen Namen. Mit folgender Konstruktion können wir aber doch Namen für unsere Funktionen verwenden:

$$(\lambda\ \mathsf{id}\ .\quad \ldots\quad)\ (\lambda\ \mathsf{x}\ .\ \mathsf{x}\ )$$

Jetzt ist im Funktionskörper unter dem Namen id die Funktion $\lambda\ \mathsf{x}\ .\ \mathsf{x}$ bekannt.

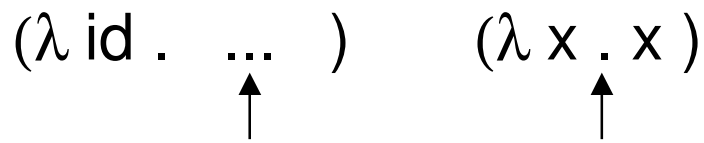
Rekursion

Von Rekursion spricht man, wenn in der Definition einer Funktion auf die Funktion selbst bezug genommen wird.

Beispiel ist die rekursive Definition der Fakultätsfunktion:

fak n := if (n = 0) then 1 else n * fak (n-1)

Dies ist so im λ-Kalkül noch nicht möglich:
Geben wir einer Funktion, wie auf der letzten Folie gesehen, einen Namen, so können wir diesen bei der Definition der Funktion noch nicht benutzten.



Rekursion (2)

Eine rekursive Funktion ist eine ganz normale Funktion, nur das in ihrem Funktionskörper neben den anderen schon definierten Bezeichnern eine weitere Funktion, nämlich sie selbst, bekannt ist.

Wir betrachten jetzt statt der Fakultätsfunktion fak die Funktion fak', die sehr ähnlich aussieht:

```
fak      n :=  if  (n = 0)  then  1  else   n *  fak (n-1)
fak' f   n :=  if  (n = 0)  then  1  else   n *  f (n-1)
```

fak' erwartet ein Parameter mehr. Übergeben wir ihr fak, so gilt:

fak' (fak) = fak

Rekursion (3)

Kommen wir zurück in das λ-Kalkül. fak' ist nicht rekursiv, und solche Funktionen können wir hier problemlos definieren.

fak' f n := if (n = 0) then 1 else n * f (n-1)
entspricht
λ f n . (...)

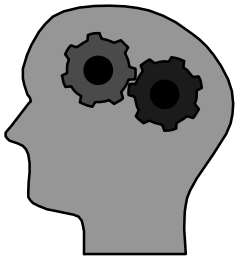
Eine Funktion **Y, die aus fak' fak** macht (ohne fak zu kennen) fehlt uns aber noch. Hätten wir sie, wären unsere Probleme gelöst.

Wir schreiben auch auf den folgenden Folien fak und fak', obwohl für eine beliebige rekursive Funktion h entsprechend h' konstruiert werden kann, und sich das folgende auf beliebige solche Paare bezieht.

Rekursion (4)

Eine Interpretation von Y:

Da folgende Gleichung gilt: fak' (fak) = fak,



kann fak als Fixpunkt von fak' gesehen werden.

Y, das fak' bekommt und fak liefert, findet also einen Fixpunkt von fak'.

Deshalb heißt Y auch Fixpunktkombinator.

Rekursion (5)

Für diese Funktion, Y würde gelten:

fak = Y (fak') = fak' (fak) = fak' (Y fak')

Für Y soll also insbesondere gelten:

Y (fak') = fak' (Y fak')

Dies ist eine rekursive Definition von Y und daher für uns unbrauchbar.

Es gibt aber tatsächliche eine nicht-rekursive Funktion, die diese Gleichung erfüllt!

Diese Funktion ist in der Tat das gesuchte Y, obwohl sie bisher nur eine notwendige Bedingung erfüllt. Der komplexe Beweis läßt sich z.B. bei *Meyer* oder *Stoy* nachlesen.

Rekursion (6)

Die Funktion **Y** wird definiert als:

$(\lambda\, h\, .\quad (\lambda\, x\, .\, h\, (x\, x))\;\; (\lambda\, x\, .\, h\, (x\, x))\quad)$

Dann ist

$$
\begin{array}{lll}
 & \mathbf{Y} & \mathbf{H'} \\
= & (\lambda\, h\, . & (\lambda\, x\, .\;\; h\, (x\, x))\;\; (\lambda\, x\, .\, h\, (x\, x))\quad)\quad H' \\
\rightarrow & & (\lambda\, x\, .\;\; H'\, (x\, x))\;\; (\lambda\, x\, .\, H'\, (x\, x)) \\
\rightarrow & & \qquad H'\, ((\lambda\, x\, .\, H'\, (x\, x))\;\; (\lambda\, x\, .\, H'\, (x\, x))) \\
\mathbf{=} & \mathbf{H'\, (Y\, H')} &
\end{array}
$$

q.e.d.

Rekursion (7)

Nun, da wir Y gefunden haben, sind unsere Probleme gelöst.

Um eine rekursive Funktion H zu definieren, betrachten wir statt dessen die nicht rekursive Funktion H'.

Genau die gleiche Funktionalität wie H liefert dann
Y (H').

Dies geht, wie wir gesehen haben, auch im λ-Kalkül.

4. Zusammenfassung

Wir haben

- die Bausteine des Lambda-Kalküls, Funktionsabstraktion und Funktionsapplikation, vorgestellt.
- als Umwandlungsregeln α- und β-Konversion kennengelernt
- formal eine Ersetzungsfunktion definiert
- einige Features realer Programmiersprachen auch im λ-Kalkül wiedergefunden, wie Zahlen, Arithmetik, Logik, Rekursion.

Ausblick

Wie wir gesehen haben, sind trotz der Schlichtheit unseres Kalküls die Fähigkeiten für komplexe Berechnungen auch hier vorhanden.

Genauer gesagt:

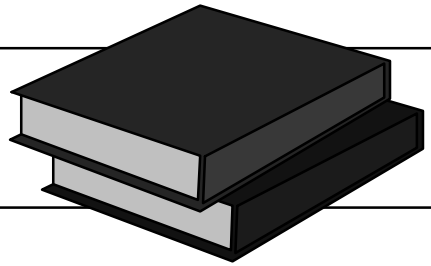
Jede von einem Computer berechenbare Funktion ist auch im Lambda-Kalkül berechenbar.

Dadurch können wir Aussagen über alle Funktionen machen, indem wir sie in unserem Kalkül analysieren. Eine typische Frage ist:

„Gibt es ein Computerprogramm, das für eine beliebige Funktion entscheidet, ob diese terminiert?“

Näheres findet sich in der Literatur.

Literatur



- Sehr formal und knapp, weitreichend:

  Barendregt, *Functional Programming and Lambda Calculus*, in: *Leeuwen*, *Handbook of Theoretical Computer Science*

- Von uns nicht gesichtet, aber in Büchern empfohlen:

  Barendregt, The Lambda Calculus - It's Syntax and Semantics

  Stoy, Denotional Semantics

- Einführung zum Thema und auch in andere Gebiete der theoretischen Informatik:

  Meyer, Introduction to the Theory of Programming Languages

- Über die Implementation funktionaler Sprachen:

  Peyton Jones, The Implementation of Functional Prog. Lang.