Der untypisierte Lambda-Kalkül

Fabian Immler

08.05.2008

1 Motivation und Einführung

Der Lambda-Kalkül ist ein formales System zur Beschreibung von Funktionen, welches Funktionsdefinition sowie das Auswerten von Funktionen umfasst. Er basiert auf Umformungen von Termen, was der „Berechnung" eines Ausdrucks entspricht. Trotz seines einfachen Aufbaus ist der Lambda-Kalkül mächtig genug alle berechenbaren Funktionen zu kodieren.

In dieser Arbeit wird zunächst auf die genaue Definition der Syntax des Lambda-Kalküls eingegangen, dann wird die Auswertung von Funktionen – sogenannte β-Reduktion – vorgestellt. Schliesslich wird gezeigt, dass die berechenbaren Funktionen im Lambda-Kalkül kodiert werden können.

2 Syntax

2.1 Terme

Definition 1 *Der Lambda-Kalkül besteht aus Termen t, die wie folgt aufgebaut sind*

$$t := x \mid (\lambda x.t) \mid (t_1\ t_2)$$

Gemeint ist damit folgendes:

$x \ldots$ **Variable**

$(\lambda x.t)$ **Abstraktion:** Definition einer Funktion mit formalem Parameter x und Körper t (t ist ein Term in dem x vorkommen kann)

$(t_1\ t_2)$ **Applikation:** Anwendung der Funktion t_1 auf den Parameter t_2

2.1.1 Beispiel: Bildung von λ-Termen

Um eine grobe Vorstellung zu vermitteln wird ein λ-Term schrittweise aufgebaut und die Bedeutung der entstandenen Terme erläutert.

- Die Variablen x und y formen den Term: $(x\ y)$

- Dieser Term wird nun als Körper einer Funktion F mit formalem Parameter x verwendet: $F := \lambda x.(x\ y)$
- Hiermit wird beschrieben, diese Funktion auf c anzuwenden: $Fc = (\lambda x.(xy))c$
- Bei der Funktionsauswertung (β-Reduktion) wird jedes x im Funktionskörper durch c ersetzt: $Fc = (\lambda x.(x\ y))c \rightarrow_\beta (c\ y)$

2.2 Schreibweise

Da Terme durch zu viele Klammern unübersichtlich erscheinen gibt es folgende Regeln um Klammern und unnötige Symbole zu vermeiden:

Variablen werden nach λ aufgelistet: $\lambda x_1 x_2 \ldots x_n.t \equiv \lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$

Applikation ist linksassoziativ $(t_1\ t_2\ t_3 \ldots t_n) \equiv (\ldots((t_1\ t_2)t_3)\ldots)t_n)$

λ bindet so weit rechts wie möglich $\lambda x.xx \equiv \lambda x.(xx) \neq (\lambda x.x)x$

Weglassen äußerer Klammern $t_1 t_2 \ldots t_n \equiv (t_1\ t_2\ \ldots\ t_n)$

2.3 Mehrstellige Funktionen

Nach der Definition einer Funktionsabstraktion ($\lambda x.t$) scheint der Lambda-Kalkül auf einstellige Funktionen beschränkt zu sein, da Terme wie $(\lambda(x, y).x+ y)$ nicht erlaubt sind.

Die Beschränkung auf einstellige Funktionen ist allerdings (wie Schönfinkel entdeckte) keine Einschränkung. Für $g(x, y) = x + y$ schreibt man im Lambda-Kalkül $g' = \lambda x.\lambda y.x + y$.

Anwendung des ersten Arguments x bildet auf eine Funktion ab, die das zweite Argument y erwartet um das Ergebnis zurückzuliefern: $g' : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ statt $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

Demnach ist $g'4$ beispielsweise wohldefiniert – das Ergebnis ist eine einstellige Funktion, die auf ihr Argument 4 addiert.

2.4 Bindungsbereich von Variablen

Prinzipiell wird zwischen freien und gebundenen Variablen unterschieden. Grundlegende Überlegung ist, dass die Variable x in $\lambda x.t$ im Term t gebunden ist.

Anhand des Aufbaus von λ-Termen kann allgemein die Menge der freien Variablen eines Terms als Funktion definiert werden:

FV: $Term \rightarrow Menge\ von\ Variablen$

- $FV(x) = x$
- $FV(s\ t) = FV(s) \cup FV(t)$

- $FV(\lambda x.t) = FV(t) \setminus x$

Ein Term ohne freie Variablen heißt *geschlossen.*

2.4.1 Beispiele

- $\lambda y.xy$: x ist frei, y gebunden
- $x(\lambda xy.z)y$: x, y und z sind frei

2.5 Kontexte

Um Terme unabhängig von ihrer Umgebung behandeln zu können ist das Konzept eines *Kontexts* nützlich:

$C[t]$ heisst *Kontext eines Terms* t falls er t als Subterm enthält.

2.5.1 Beispiel

$$(\lambda x.x)(\lambda y.t\ y)u = C[\lambda y.t\ y]$$

2.6 Substitution

Substitution bildet die Grundlage für β-Reduktion – das Auswerten von Funktionsanwendungen.

2.6.1 Notation

$s[t/x]$... sprich ,,s mit t für x" bedeutet, dass im Term s jedes Vorkommen der Variablen x durch den Term t ersetzt wird.

2.6.2 Beispiele

- $(lxmbdx)[a/x] = lambda$
- $(\lambda x.yx)[(\lambda z.st)/y] = \lambda x.(\lambda z.st)x$

Vorsicht! Freie Variablen dürfen durch Substitution nicht gebunden werden:

$$\lambda y.(x[y/x]) \neq \lambda y.y$$

Deshalb die Konvention, gebundene Variablen automatisch so umzubenennen, dass sie von allen freien Variablen verschieden sind:

$$\lambda y.(x[y/x]) = \lambda y'.y$$

2.7 α-Konversion

Grundlage der α-Konversion ist die Idee, dass beispielsweise die Terme $(\lambda x.xy)$ und $(\lambda z.zy)$ die gleiche Funktion beschreiben (modulo Umbenennung gebundener Variablen).

Definition 2 *Die Relation $\rightarrow_\alpha$ ist definiert als ein-Schritt-Variablenumbenennung:*

$$s \rightarrow_\alpha t \Leftrightarrow (s = C[\lambda x.u]) \wedge (t = C[\lambda y.u[y/x]]) \wedge y \notin FV(u)$$

α-Äquivalenz ist definiert als die reflexive und transitive Hülle von $\rightarrow_\alpha$:

$$s =_\alpha t \Leftrightarrow s \rightarrow_\alpha^* t$$

3 β-Reduktion

Unter β-Reduktion versteht man die Auswertung von Funktionsanwendungen im Lambda-Kalkül.

Reduziert werden können Terme der Form $(\lambda x.s)t$, genannt *β-Redex* (*red*ucible *ex*pression). Dies geschieht durch Substitution mit t für x in s.

Definition 3 *In jedem Term der einen β-Redex enthält kann dieser reduziert werden:*

$$C[(\lambda x.s)t] \rightarrow_\beta C[s[t/x]]$$

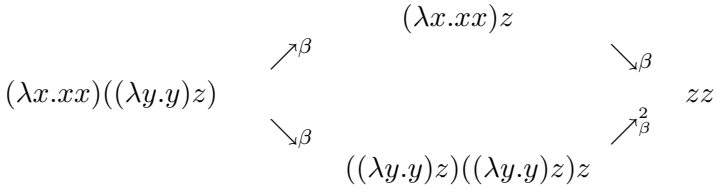
Änderungen geschehen nur am Redex, der Kontext bleibt unberührt.

Definition 4 *Ein Term ist in β-Normalform, wenn er nicht weiter reduziert werden kann.*

Somit entspricht die β-Reduktion dem ,,Berechnen" einer Funktion: $\rightarrow_\beta$ entspricht einem Berechnungsschritt, die β-Normalform ist das Ergebnis.

3.1 Eigenschaften

Da ein Term mehrere Redexe enthalten kann, kann β-Reduktion verschiedene Wege einschlagen, was an untenstehendem Beispiel nachvollzogen werden kann.



Hier entsteht bei beiden Wegen dieselbe β-Normalform. Manche Terme besitzen keine β-Normalform wie etwa $\Omega := (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \Omega$.

Für den Lambda-Kalkül als Berechnungsmodell sind folgende Fragen von großer Bedeutung:

- Kann man auf verschiedenen Rechenwegen unterschiedliche Ergebnisse (β-Normalformen) erhalten? (*Nicht erwünscht!*)
- Wenn ein Term eine β-Normalform besitzt: existiert eine Strategie, mit der diese sicher erreicht wird? (*Erwünscht!*)

3.1.1 Konfluenz

Die Relation $\rightarrow_\beta$ ist konfluent. Das heißt im Lambda-Kalkül besitzt ein Term nur maximal eine β-Normalform, was ein Ergebnis des *Church-Rosser-Theorems* ist. Falls ein Term auf verschiedene Arten reduziert werden kann, führt weitere Reduktion auf den selben Term:

$$\forall r, s_1, s_2.r \rightarrow_\beta^* s_1 \wedge r \rightarrow_\beta^* s_2 \exists t.s_1 \rightarrow_\beta^* t \wedge s_2 \rightarrow_\beta^* t$$

3.1.2 Reduktionsstrategien

Unterscheidung verschiedener Strategien:

call-by-name Reduzierung des jeweils linkesten β-Redexes.

call-by-value Reduzierung des innersten β-Redexes.

Aus praktischer Sicht erscheint die *call-by-value*-Strategie im Vorteil, da Argumente nur einmal ausgewertet werden müssen:

$$(\lambda x.xxx \ldots x)((\lambda y.s)t)$$

Call-by-name hat den Vorteil, dass nicht benötigte Argumente nicht ausgewertet werden, *call-by-value* geriete hier sogar in eine Endlosschleife:

$$(\lambda x.a)(\Omega)$$

Theorem 1 *Wenn t eine β-Normalform hat, dann wird diese auch erreicht, wenn immer der jeweils linkeste $\beta-$Redex reduziert wird.*

4 Kodierung der berechenbaren Funktionen im Lambda-Kalkül

In diesem Abschnitt wird vorgestellt wie einige Konzepte die für eine ,,vollwertige" Programmiersprache nötig sind im Lambda-Kalkül umgesetzt werden können.

Dazu gehören verschiedene Datentypen – natürliche Zahlen, boolesche Werte, Paare – die im Lambda-Kalkül beschrieben werden, ebenso Fallunterscheidungen und Rekursion.

Damit soll die Intuition vermittelt werden, dass der Lambda-Kalkül Turing-vollständig ist.

4.1 Datentypen

4.1.1 Boolesche Werte

Gesucht sind λ-Terme $if\ true$ und $false$, sodass gilt:

$$if\ true\ x\ y \rightarrow_\beta^* x$$
$$if\ false\ x\ y \rightarrow_\beta^* y$$

Sei $if := (\lambda cte.cte)$, dann folgt $if\ true\ x\ y \rightarrow_\beta true\ x\ y$.

Demnach muss $true$ als eine Funktion definiert werden, die zwei Argumente erhält und das erste unverändert zurückgibt:

$$true := (\lambda ab.a)$$

Analog die Definintion von $false$:

$$false := (\lambda ab.b)$$

Damit gelingt die Fallunterscheidung:

$if\ true\ x\ y$
$= (\lambda cte.cte) true\ x\ y \rightarrow_\beta true\ x\ y$
$= (\lambda ab.a) xy \rightarrow_\beta x$

4.2 Paare

Gesucht sind λ-Terme fst, snd und $pair$ mit folgenden Eigenschaften:

$$fst(\ pair\ x\ y) \rightarrow_\beta^* x$$
$$snd(\ pair\ x\ y) \rightarrow_\beta^* y$$

Erfüllt werden sie durch diese Terme:

$$fst = \lambda p.p\ true$$
$$snd = \lambda p.p\ false$$
$$pair = \lambda xy.\lambda z.zxy$$

Beispiel:

$snd(pair\ x\ y)$
$= snd((\lambda xyz.zxy)xy) \rightarrow_\beta snd(\lambda z.zxy)$
$= (\lambda p.p\ false)(\lambda z.zxy) \rightarrow_\beta (\lambda z.zxy) false \rightarrow_\beta false\ xy \rightarrow_\beta y$

4.3 Natürliche Zahlen (Church Numerale)

Die Kodierung nach Alonzo Church, dem ,,Erfinder" des Lambda-Kalküls basiert auf der Idee, Zahlen als geschlossene Terme auf Grundlage einer festen Null und einer Nachfolgerfunktion zu definieren:

$0 := \lambda f.\lambda x.x$
$1 := \lambda f.\lambda x.fx$
$2 := \lambda f.\lambda x.f(fx)$
$3 := \lambda f.\lambda x.f(f(fx))$
$\vdots$
n:= $\lambda f.\lambda x.f^n(x) = \lambda f.\lambda x.f(f(\ldots(f\ x)\ldots))$

Wichtig ist für weitere Überlegungen, dass die Church-Kodierung einer Zahl n als zweistellige Funktion interpretiert werden kann, welche die als erstes Argument übergebene Funktion f sukzessive n mal auf den Parameter x anwendet:

$$(n\ f\ x) \rightarrow_\beta f^n(x)$$

Um mit den Church-kodierten Zahlen sinnvoll rechnen zu können sind beispielsweise Funktionen wie *succ* mit $succ\ n = n + 1$, *add* mit $add\ n\ m = n + m$ oder ein Test auf Null *iszero* nötig.

4.3.1 Nachfolgerfunktion

$$succ := \lambda n.\lambda f x.f(nfx)$$

nfx ist die Darstellung der Zahl n mit f als Nachfolgerfunktion und x als 0 – also $f^n(x)$). Mit einem um diesen Term geklammerten f wird die Repräsentation der um eins erhöhten Zahl erstellt.

$$succ\ n = (\lambda n.\lambda f x.f(nfx))n \rightarrow_\beta \lambda f x.f(nfx)$$

$$\rightarrow_\beta \lambda f x.f(f^n(x)) = \lambda f x.f^{n+1}(x)) = n + 1$$

4.3.2 Addition

$$add := \lambda m n.\lambda f x.m\ f\ (nfx)$$

Ein m-maliges Anwenden der Funktion f auf (nfx); um die Darstellung von n wird noch m-mal das f geklammert:

$$add\ m\ n = \lambda f x.\underline{m\ f\ (n\ f\ x)} \rightarrow_\beta \lambda f x.f^m(\underline{n\ f\ x})$$

$$\rightarrow_\beta \lambda f x.f^m(f^n(x)) = \lambda f x.f^{m+n}(x) = m + n$$

$$add\ 2\ 3 = \lambda f x.\underline{2f(3fx)} \rightarrow_\beta \lambda f x.f(f(\underline{3fx})) \rightarrow_\beta \lambda f x.f(f(\ f(f(f\ x))\ ))$$

4.3.3 Test auf Null

$$iszero := \lambda n.n(\lambda x.false)true$$

Ein Test auf Null ist notwendig, um Fallunterscheidungen nach Rechenergebnissen ausführen zu können.

$$iszero\ 0 \rightarrow_\beta (\lambda fx.x)(\lambda x.false)true \rightarrow_\beta true$$

$$\begin{aligned} iszero\ 2 &\rightarrow_\beta (\lambda fx.f(fx))(\lambda x.false)true \\ &\rightarrow_\beta (\lambda x.false)((\lambda x.false)true) \rightarrow_\beta^* false \end{aligned}$$

4.3.4 Multiplikation

$$mult := \lambda mn.\lambda fx.m(nf)x$$

$$\begin{aligned} mult\ m\ n = \lambda fx.m(nf)x &\rightarrow_\beta \lambda fx.(nf)^m(x) \\ &\rightarrow_\beta \lambda fx.(f^n)^m(x) = \lambda fx.f^{n*m}(x) = n * m \end{aligned}$$

4.3.5 Vorgängerfunktion

Die Bildung einer Vorgängerfunktion *pred* ist im Vergleich zur Nachfolgerfunktion *succ* trickreicher. Zunächst wird eine Funktion *pairsucc* definiert mit $pairsucc(pair(x, n)) = pair(n, n + 1)$

$$pairsucc = \lambda p.(snd\ p, succ(snd\ p))$$

Nach n-maligem Anwenden dieser Funktion auf das Paar (0,0) erhält man das Paar (n-1,n):

$$pairsucc^n(pair(0, 0)) = pair(n - 1, n)$$

Dies motiviert folgende Definition von *pred*: $pred(n) = fst(pairsucc^n(pair(0, 0)))$, im Lambda-Kalkül geschrieben:

$$pred = \lambda n.fst(n\ pairsucc\ pair(0, 0))$$

4.4 Rekursive Funktionen

Im Lambda-Kalkül können Funktionen nicht auf sich selbst zugreifen. Eine rekursive Funktion f muss demnach nicht-rekursiv dargestellt werden.

Gehen wir von einer rekursiven Beschreibung einer Funktion f aus:

$$f\ x = t \quad \text{mit} \quad t = C[f]$$

Im Funktionskörper t muss x also gebunden sein:

$$f = \lambda x.t = \lambda x.C[f]$$

Definiert wird nun folgende Funktion, die starke Ähnlichkeit mit f hat:

$$F := \lambda f.\lambda x.t = \lambda f.\lambda x.C[f]$$

Hier ist f als formaler Parameter gebunden. F ist demnach nicht rekursiv. f hat hier schon nichts mehr mit der obigen rekursiven Funktion zu tun, ist nur noch formaler Parameter und könnte ebensogut e,g oder h heißen.

Auffallend ist folgendes:

$$F\ f = f$$

f ist also Fixpunkt der Funktion F. Gäbe es einen Fixpunktoperator fix der von jeder Funktion einen Fixpunkt bestimmen kann – also $\forall f.fix\ f = f\ (fix\ f)$ – dann könnte f nichtrekursiv definiert werden als:

$$f = fix\ F$$

4.4.1 Beispiel

Die rekursive Fakultätsfunktion:

$$fak\ n = if(iszero\ n)1(mult\ n\ (fak(pred\ n)))$$

Die unzuläßige Darstellung mit gebundenem n im Lambda-Kalkül($fak = \lambda n.C[fak]$):

$$fak = \lambda n.if(iszero\ n)1(mult\ n\ (fak(pred\ n)))$$

Definition der nichtrekursiven Funktion FAK ($FAK = \lambda f.\lambda n.C[f]$):

$$FAK = \lambda f.\lambda n.if(iszero\ n)1(mult\ n\ (f(pred\ n)))$$

Nichtrekursive Darstellung von fak:

$$fak = fix\ FAK$$

Unter Ausnutzung der Eigenschaften des Fixpunktoperators fix kann die Fakultät berechnet werden:

$$\begin{aligned}
fak\ 2 &= (fix\ FAK)2 = FAK(fix\ FAK)2 \\
&= if(iszero\ 2)1(mult\ 2\ ((fix\ FAK)(pred\ 2))) \\
&\rightarrow_\beta mult\ 2\ (fix\ FAK)(pred\ 2) \rightarrow_\beta mult\ 2\ ((fix\ FAK)1) \\
&\rightarrow_\beta mult\ 2\ if(iszero\ 1)1(mult\ 1\ ((fix\ FAK)(pred\ 1))) \\
&\rightarrow_\beta mult\ 2\ (mult\ 1\ ((fix\ FAK)(pred\ 1))) \\
&\rightarrow_\beta mult\ 2\ (mult\ 1\ (FAK(fix\ FAK)0)) \\
&\rightarrow_\beta mult\ 2\ (mult\ 1\ (if(iszero\ 0)1(mult\ 0\ ((fix\ FAK)(pred\ 0))))) \\
&\rightarrow_\beta mult\ 2\ (mult\ 1\ 1)
\end{aligned}$$

4.4.2 Der Fixpunktoperator Y

Bleibt noch die Existenz eines Fixpunktoperators fix zu zeigen: Für Church's Operator Y muss das Ziel sein $Y\ t = t(Y\ t)$.

Definiert wird V_f:

$$V_f = \lambda x.f(xx)$$

Und mit $Y = \lambda f.V_f V_f$ erhält man einen Operator mit den gewünschten Eigenschaften:

$$Yt \rightarrow_\beta V_t V_t = (\lambda x.t(xx))V_t = t(V_t V_t) \ {}_\beta \leftarrow t((\lambda f.V_f V_f)t)$$

4.5 Lambda-Definierbarkeit

Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt λ-definierbar, falls es einen Term t gibt mit

- $(t\ x) \rightarrow_\beta^* y$,falls $f(x) = y$
- $(t\ x)$ hat keine Normalform, falls $f(x)$ undefiniert

Theorem 2 *Alle (Turing-, Registermaschinen-, ...)berechenbaren Funktionen sind λ-definierbar und umgekehrt.*

Literatur

[1] Chris Hankin. *An Introduction to Lambda-Calculi for Computer Scientists.* King's College Publications, 2004.

[2] Prof. Tobias Nipkow. Lambda-Kalkül. *http://www4.in.tum.de/lehre/vorlesungen/logik/WS0607/lambda.pdf*, 2004.