Proseminar: Programmiersprachen
bei Prof. Tobias Nipkow

Der untypisierte Lambda-Kalkül als Programmiersprache

von
Benjamin Peherstorfer
betreut von
Prof. Tobias Nipkow

23. November 2006

Inhaltsverzeichnis

1 Grundlagen des untypisierten Lambda-Kalküls 3
1.1 Definition der Terme . . . 3
1.2 Notation . . . 3
1.3 Gültigkeitsbereich von Variablen . . . 3
1.4 Substitution . . . 4
1.5 α-Konversion . . . 5
1.6 β-Reduktion . . . 6
1.6.1 β-Redex . . . 6
1.6.2 β-Reduktion . . . 6
1.6.3 β-Normalform . . . 7
1.6.4 Eigenschaften der β-Reduktion . . . 7
1.7 Reduktionsstrategien . . . 8
1.8 Currying . . . 8

2 Lambda-Kalkül als Programmiersprache 9
2.1 Church Booleans . . . 9
2.2 Paare und Listen . . . 9
2.3 Church Numerale . . . 10
2.4 Rekursive Funktionen . . . 13
2.4.1 Fixpunkte . . . 13
2.4.2 Darstellung rekursiver Funktionen . . . 13
2.4.3 Beispiel . . . 14
2.4.4 Berechenbare Funktionen auf $\mathbb{N}$ . . . 14

1 Grundlagen des untypisierten Lambda-Kalküls

1.1 Definition der Terme

Defintion (Terme des Lambda-Kalküls) In seiner reinsten Form existieren im Lambda-Kalkül nur drei Sorten von Termen die wie folgt festgelegt werden können:

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

wobei x Element aus einer mindestens abzählbar-unendlichen Menge von Variablen ist. Der Term $(\lambda x.t)$ wird als *Abstraktion* und der Term $(t_1 \;\; t_2)$ als *Applikation* bezeichnet.
Die Abstraktion repräsentiert die Funktion, d.h. die Variable x wird als Paramter aufgefasst, während t den Funktionsrumpf darstellt. Die Applikation ist der Funktionsaufruf mit Funktion t_1 und Argument t_2.

Beispiele Identitätsfunktion: $(\lambda x.x)$. Eine Funktion die immer die Identitätsfunktion zurück gibt: $(\lambda x.(\lambda y.y))$.

Von dieser Defintion ausgehend können nun Methoden für den Umgang mit Lambda-Termen entwickelt werden. Dies soll in den restlichen Abschnitten dieses Kapitels geschehen.

1.2 Notation

Konventionen Im Allgemeinen werden a, b, c, x, y, z für Variablen und l, m, n, s, t, u, v, w für Lambda-Terme verwendet. Im zweiten Teil dieser Arbeit, wo der Lambda-Kalkül als "Programmiersprache" verwendet wird, mögen Abweichungen auftreten.

Effiziente Klammerung Die Definition der Lambda-Terme bringt exzessive Klammerung mit sich. Die meisten können jedoch durch folgende Vereinbarungen eingespart werden.

a) Die Applikation ist *linksassoziativ*, d.h. $(t_1 t_2 \ldots t_n) \equiv (\ldots(t_1 t_2)t_3)\ldots t_n)$

b) Die Abstraktion ist *rechtsassoziativ*, sie bindet so weit nach rechts wie möglich: $(\lambda x.(\lambda y.(xy))) \equiv (\lambda x.\lambda y.xy)$.

c) Innere Punkte und Lambdas können weggelassen werden: $\lambda x_1 x_2 x_3 \ldots x_n.t \equiv \lambda x_1.\lambda x_2.\ldots\lambda x_n.t$

d) Äußere Klammern werden weggelassen:$(\lambda x.t) \equiv \lambda x.t$ und $(t_1 \ldots t_n) \equiv t_1 \ldots t_n$.

Beispiele

(i) $\lambda xyz.zyx \equiv (\lambda x.(\lambda y.(\lambda z.((zy)x))))$

(ii) $\lambda xx.x(xx) \equiv (\lambda x.(\lambda x.(x(xx))))$

(iii) $\lambda xy.x \equiv (\lambda x.(\lambda y.(x)))$

1.3 Gültigkeitsbereich von Variablen

Dieser Abschnitt soll den Gültigkeitsbereich (*scope*) einer Variable klären.

(Informelle) Definition Eine Variable x wird durch eine Abstraktion $\lambda x.t$ *gebunden* wenn sie im Term t dieser Abstraktion vorkommt. Andernfalls ist sie *frei*. Wichtig dabei ist, dass eine Variable x durch das erste λx oberhalb von x gebunden wird.

Beispiele

(i) $\lambda x.x$, x ist gebunden

(ii) $\lambda y.xy$, x ist frei, y ist gebunden

(iii) $\lambda x.xx$, x ist gebunden

(iv) $(\lambda xy.xy)x$, x in () ist gebunden, das äußere rechte x nicht. Die Variable y ist gebunden

(v) $x\lambda z.(x\lambda a.azy)$, x, y sind frei, z, a gebunden.

Formale Definition Formal kann die Menge der freien Variablen als Funktion definiert werden.

$$
\begin{array}{lcl}
FV : \text{Term} & \rightarrow & \text{Menge von Variablen} \\
FV(x) & = & \{x\} \\
FV(s\ \ t) & = & FV(s) \cup FV(t) \\
FV(\lambda x.t) & = & FV(t) \backslash \{x\}
\end{array}
$$

Kombinator Besitzt ein Term t keine freien Variablen wird er *Kombinator* genannt. Der Term ist dann *geschlossen*.

1.4 Substitution

Schreibweise: Die Substitution von t für x in s wird als $s[t/x]$ geschrieben und als "s mit t für x" gesprochen. Bei der Substitution wird klar, dass die Unterscheidung zwischen freien und gebundenen Variablen äußerst wichtig ist. Probleme treten auf wenn man versucht (a)eine gebundene Variable zu ersetzen oder (b)durch Ersetzungen, freie Variablen plötzlich zu gebundenen werden.

Beispiele

(i) Zu (a): ersetze Variable x durch Variable y in $\lambda x.x$. Ergebnis: $\lambda x.y$. Dies entspricht jedoch nicht mehr der Identität und ist daher falsch!

(ii) Zu (b): Man geht von dem Term $\lambda x.y$ aus und ersetzt die freie Variable y durch x und erhält die Identität $\lambda x.x$. Aber $\lambda x.y \neq \lambda x.x$.

Es gibt mehrere Ansätze die Substitution zu definieren. Hier wird nur der ursprüngliche Ansatz von A. Church verfolgt, für weitere wird auf [Han04] verwiesen.

Definition Die Subsitution wird rekursiv definiert durch:

$$
\begin{array}{lrcll}
(1) & x[t/x] & = & t & \\
(2) & y[t/x] & = & y & \text{falls } y \neq x \\
(3) & (s_1\ \ s_2)[t/x] & = & (s_1[t/x])(s_2[t/x]) & \\
(4) & (\lambda x.s)[t/x] & = & \lambda x.s & \\
(5) & (\lambda y.s)[t/x] & = & \lambda y.(s[t/x]) & \text{falls } x \neq y \wedge y \notin FV(t) \\
(6) & (\lambda y.s)[t/x] & = & \lambda z.(s[z/y][t/x]) & \text{falls } x \neq y \wedge z \notin FV(t) \cup FV(s)
\end{array}
$$

Anmerkungen zur Definition

(i) zu (4): hier muss x gebunden sein, da λx ganz außen steht und damit alle x in s bindet. Es darf nicht substituiert werden.

(ii) zu(5): Diese Regel darf nur ausgeführt werden wenn y in t gebunden ist. Es können daher keine neuen Bindungen entstehen da es kein freies y in t gibt welches von λy gebunden werden könnte.

(iii) zu(6): Ist y in t frei, so könnte eine neue Bindung entsehen. Um dies zu verhindern wird die Variable y in z umbenannt bevor x durch t substituiert wird.

Beispiele

(i) $\lambda x.yx[x/y] \to_6 \lambda z.xz$

(ii) $(x\ \ \lambda x.x)[y/x] \to_3 (x[y/x]\ \ (\lambda x.x)[y/x]) \to_{1,4} (y\ \ \lambda x.x)$

(iii) $(\lambda zy.xx)[y/x] \to_5 \lambda z.((\lambda y.xx)[y/x]) \to_6 \lambda z.(\lambda z.((xx)[z/y][y/x]))$
$\to_3 \lambda z.(\lambda z.(((x)[z/y][y/x])\ \ (x)[z/y][y/x])) \to_1 \lambda z.(\lambda z.(((x)[y/x]\ \ (x)[y/x])))$
$\to_1 \lambda z.(\lambda z.yy) \to \lambda zz.yy$

(iv) $((\lambda x.zx)(\lambda z.zx))[(yx)/z] \to_3 ((\lambda x.zx)[(yx)/z](\lambda z.zx)[(yx)/z]) \to_{6,4}$
$\to_{6,4} (\lambda z_1.((zx)[z_1/x][(yx)/z]))(\lambda z.zx) \to (\lambda z_1.yxz_1)(\lambda z.zx)$

1.5 α-Konversion

Wie intuitiv klar ist, stehen $\lambda x.x$ und $\lambda y.y$ jeweils für die Identität. Die Substitution wie oben definiert, lässt jedoch solche Änderungen nicht zu. Wann gebundene Variablen in einem Term unbenannt werden dürfen und wie das geschieht, stellt die Definition der α-Konversion klar:

Definition Die Definition der α-Konversion lautet:

$$\lambda x.s \to_\alpha \lambda y.(s[y/x]) \qquad \text{falls } y \notin FV(s).$$

Die gebundene Variable x darf durch y ersetzt werden, wenn y in s nur gebunden oder gar nicht vorkommt. Die Variable y darf nicht frei vorkommen, sonst würde gelten $\lambda x.xy \to_\alpha \lambda y.yy$.

Beispiele

(i) $\lambda xy.yx \to_\alpha \lambda zy.yz \to_\alpha \lambda zx.xz \to_\alpha \lambda yx.xy$ man schreibt auch kurz $\lambda xy.yx =_\alpha \lambda yx.xy$.

(ii) $\lambda x.(\lambda x.x)x \to_\alpha \lambda y.(\lambda y.y)y \to \lambda y.((\lambda y.y) \to_\alpha (\lambda x.x))y \to \lambda y.(\lambda x.x)y$ (Es wurde $(\lambda x.x)$ als Teilterm behandelt!)

(iii) $\lambda x.xy \not\to_\alpha$ aber $\lambda x.xz \to_\alpha \lambda y.yz$.

Mit der α-Konverison können nun die Regeln für die Subsitution stark vereinfach werden, falls man noch die folgende Konvention einhält:

Automatisches Umbenennen Damit gebundene Variablen von freien Variablen verschieden sind, werden sie automatisch unbenannt. Dies ist möglich, da (wie in der Definition vorausgesetzt) die Menge der Variablen mindestens abzählbar-unendlich ist. Es wird dadurch vermieden, neue Bindungen zu erschaffen.
Das folgende Beispiel soll diese Konvention veranschaulichen: $\lambda x.\lambda y.x \rightarrow_\alpha \lambda y.\lambda z.y$.

Vereinfachung der Substitution Regeln (4)-(6) der Substitution fallen weg und werden durch

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

ersetzt. Durch die obige Konvention werden gebundene Variablen umbenannt, sollten sie mit freien Variablen kollidieren und so können keine neuen, unerwünschten Bindungen bei der Substitution entstehen. Genau das hätten aber Regel (5) und (6) verhindert und damit sind sie überflüssig.

1.6 β-Reduktion

Die β-Reduktion bietet eine Möglichkeit Lambda-Terme auszuwerten. Bevor jedoch die β-Reduktion selbst definiert wird, muss noch die Gestalt der Terme identifiziert werden, welche man mit der β-Reduktion reduzieren kann.

1.6.1 β-Redex

Definition Einen Lambda-Term der Form

$$(\lambda x.s)t$$

nennt man β-Redex (reducible expression). Ein β-Redex besteht also aus einer Abstraktion gefolgt von einem beliebigen Term t.

1.6.2 β-Reduktion

Definition Die β-Reduktion ist folgend definiert

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

Es wird t für den formalen Parameter x in s substituiert. Die β-Reduktion ist die wichtigste Regel zum Auswerten von Lambda-Termen. Werden von einem Term zum Ziel-Term mehrere Schritte benötigt, schreibt man auch kurz $t_1 \rightarrow_\beta^* t_2$. Es bezeichnet damit $\rightarrow_\beta^*$ die reflexive, transitive Hülle von $\rightarrow_\beta$.

Die Auswertung von Lambda-Termen besteht aus einer Folge von β-Reduktionen, möglicherweise unterbrochen von mehreren α-Konversionen.

Beispiele

(i) $(\lambda x.y)z \rightarrow_\beta y$

(ii) $((\lambda xy.y)z)a \rightarrow_\beta (\lambda y.y)a \rightarrow_\beta a$

(iii) $(\lambda xy.xyx)y \rightarrow_\alpha (\lambda xy'.xy'x)y \rightarrow_\beta \lambda y'.yy'y$

(iv) $(\lambda y.yz)(\lambda x.x) \rightarrow_\beta (\lambda x.x)z \rightarrow_\beta z$

(v) $(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \dots$

1.6.3 β-Normalform

Um drei Eigenschaften der β-Reduktion im nächsten Abschnitt zu zeigen, muss vorher noch die β-Normalform definiert und untersucht werden.

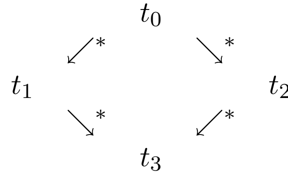
Definition Der Term t_1 ist in β-Normalform wenn kein Term t_2 mit $t_1 \rightarrow_\beta t_2$ existiert. Mit anderen Worten: Ein Term t_1 ist in β-Normalform wenn keine weiter β-Reduktion mehr möglich ist, d.h. kein β-Redex im Term als Subterm[1] existiert.

Es stellen sich zwei grundlegende Fragen: (i) Besitzt jeder Lambda-Term eine β-Normalform und (ii) ist diese Normalform eindeutig?

Frage (i) kann durch das obige Beispiel (v) sofort mit Nein beantwortet werden. Der β-Redex $(\lambda x.xx)(\lambda x.xx)$ besitzt keine β-Normalform.

Könnte man zeigen, dass die β-Reduktion konfluent ist, so würde sich auch Frage (ii) klären.

Definition Es bezeichne $\rightarrow^*$ die reflexive, transitive Hülle der Relation $\rightarrow$. Die reflexive, transitive Hülle der Relation $\rightarrow$ wird als *konfluent* bezeichnet, wenn für alle Terme t_0, t_1, t_2 mit $t_0 \rightarrow^* t_1$ und $t_0 \rightarrow^* t_2$ ein Term t_3 existiert mit $t_1 \rightarrow^* t_3$ und $t_2 \rightarrow^* t_3$. Veranschaulichung:



Betrachtet man t_1 und t_2 als zwei verschiedene β-Normalformen von t_0, so müssten sich beide Terme t_1 und t_2 zu t_3 reduzieren lassen, falls die β-Reduktion konfluent ist. Lassen sich aber t_1 und t_2 weiter reduzieren, so können sie keine β-Normalformen von t_0 sein. Es ist daher nur noch die "leere Reduktion" möglich. Dann folgt jedoch $t_1 = t_2$. Ist daher die β-Reduktion konfluent, existiert maximal eine β-Normalform eines beliebigen Terms.

Wie [Nip04] zeigt, ist die β-Reduktion konfluent und damit ist die Normalform eines Terms bzgl. der β-Reduktion eindeutig, falls sie existiert.

1.6.4 Eigenschaften der β-Reduktion

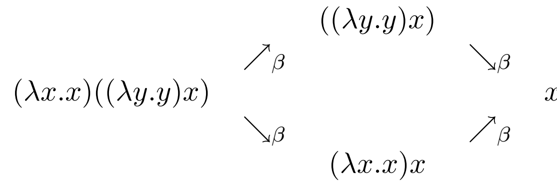
Die Haupteigenschaften der β-Reduktion sind

(i) nicht terminierend (siehe Beispiel (v) oben).

[1] Für die Definition von Subterm siehe [Nip04].

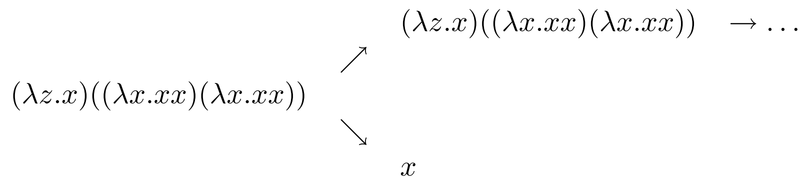
(ii) Konfluenz: jeder Term hat höchstens eine β-Normalform.

(iii) nicht-deterministisch, wie folgendes Beispiel zeigt



1.7 Reduktionsstrategien

Die β-Reduktion ist nicht-deterministisch, d.h. es führen oft mehrere Weg zur Normalform, falls sie existiert. Eine Reduktionsstrategie legt fest, welcher Term als nächstes reduziert wird. Dass dies nicht unerheblich ist, soll das nächste kurze Beispiel zeigen:



Full β-Reduction Diese Strategie erlaubt das Reduzieren jedes Redexes zu jeder Zeit.

Normal Order Strategy Es wird immer der am weitesten links stehende Redex reduziert.

Call by Name Strategy Arbeitet wie *Normal Order Strategy* nur lässt diese Auswertungsstrategie keine Reduktionen innerhalb einer Abstraktion zu. So kann z.B. $(\lambda x.((\lambda y.y)z))$ unter *Call by Name* nicht weiter reduziert werden. Unter *Normal Order Strategy* reduziert sich das Beispiel jedoch zu $\lambda x.z$.

Call by Value Strategy Eine Strategie die nur eine Reduktion von den äußersten Redexes zulässt und auch nur dann, wenn die rechte Seite des Redexes bereits so weit wie möglich reduziert wurde. Mit *Call by Value* lässt sich $(\lambda x.x)((\lambda x.x)z)$ nicht reduzieren. Das Beispiel ergibt unter *Call by Name* z.

1.8 Currying

Durch die schlichte Definition des Lambda-Kalküls, sind nur einstellige Funktionen zugelassen. Dies bringt jedoch keinen Nachteil mit sich, da jede mehrstellige Funktion durch mehrere, ineinander verschachtelte Funktionen dargestellt werden kann. Dies nennt man *Currying* (nach Haskell Curry) oder auch *Schönfinkeln* (nach Moses Schönfinkel).

Es soll die Funktion $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, (x, y) \mapsto x + y$ durch einen Lambda-Term dargestellt werden. Dazu wird eine Funktion definiert die ein Argument x entgegennimmt und als Resultat eine Funktion liefert die ein Argument y entgegennimmt und zu x addiert.
Die Funktion g wäre daher $\lambda x.(\lambda y.(x + y))$[2].

[2]Es handelt sich hier nicht um einen gültigen Lambda-Term, da das Zeichen + undefiniert ist.

2 Lambda-Kalkül als Programmiersprache

2.1 Church Booleans

Das Konstrukt einer Fallunterscheidung umfasst nicht nur die If-Abfrage selbst sondern es muss auch geklärt werden was true bzw. false eigentlich ist, d.h. welchem Term true und false entspricht.
Die Fallunterscheidung ist ein Konstrukt der Form "if t then v else w". Im Lambda-Kalkül wird die Auswahl des richtigen Zweiges nicht durch das "if" getroffen, sondern durch den Wahrheits-Term. Den true- bzw. false-Termen wird daher der then- und der else-Zweig übergeben. Die Entscheidung welcher zurückgegeben wird, liegt am Wahrheits-Term selbst. Es soll also gelten "true $vw = v$" und "false $vw = w$".

Definition Man definiert true und false daher wie folgt

$$\begin{aligned} \text{true} &\equiv \lambda x.\lambda y.x \\ \text{false} &\equiv \lambda x.\lambda y.y \end{aligned}$$

Damit man ein Konstrukt der üblichen "if t then v else w"-Form hat, muss noch ein Term erstellt werden, der die beiden Terme v und w auf den Wahrheits-Term anwendet.

Definition Ein Term test soll daher sowohl $\text{test true}\, ab = a$ als auch $\text{test false}\, ab = b$ erfüllen. Man definiert daher test wie folgt:

$$\text{test} \equiv \lambda l.\lambda m.\lambda n.lmn$$

Beispiele

(i) $\text{test true}\, vw \equiv (\lambda lmn.lmn)\,\text{true}\, vw \rightarrow_\beta (\lambda mn.\,\text{true}\, mn)vw \rightarrow^*_\beta \text{true}\, vw \equiv (\lambda x.\lambda y.x)vw \rightarrow^*_\beta v$

(ii) $\text{test false}\, vw \rightarrow^*_\beta \text{false}\, vw \rightarrow^*_\beta w$

Mit den oben definierten Wahrheits-Termen können auch die logischen Operatoren and, or und not definiert werden:

(i) $\text{and} \equiv \lambda xy.xy\,\text{false}$

(ii) $\text{or} \equiv \lambda xy.x\,\text{true}\, y$

(iii) $\text{not} \equiv \lambda x.x\,\text{false}\,\text{true}$

2.2 Paare und Listen

Ein Paar verwaltet zwei Elemente, indem auf Wunsch entweder das erste oder das zweite Element zurückgegeben wird. Es sollen daher folgende Beziehungen zwischen den drei Termen pair, fst, snd gelten:

$$\begin{aligned} \text{fst}(\text{pair}\, ab) &= a \\ \text{snd}(\text{pair}\, ab) &= b \end{aligned}$$

Motiviert durch die geforderten Eigenschaften von pair, fst und snd ergibt sich folgende Definition.

Definition Der Term pair wendet einen noch zu übergebenden Boolean auf zwei Terme an. Es wird so je nach Wahrheitswert des Boolean entweder das erste oder das zweite Element bzw. Term zurückgegeben. Die Funktionen fst und snd übergeben lediglich diesen Boolean.

$$\begin{aligned} \text{pair} &\equiv \lambda x.\lambda y.\lambda z.zxy \\ \text{fst} &\equiv \lambda p.p\,\text{true} \\ \text{snd} &\equiv \lambda p.p\,\text{false} \end{aligned}$$

Beispiel $\text{snd}\,\text{pair}\,st \rightarrow^*_\beta \text{snd}(\lambda z.zst) \rightarrow_\beta (\lambda z.zst)(\lambda xy.y) \rightarrow_\beta \text{false}\,st \rightarrow^*_\beta t.$

Verschachtelt man Paare ineinander, so entsteht eine Liste:

$$\text{pair}\,a(\text{pair}\,b(\text{pair}\,c\,\text{nil})) \equiv (\lambda z.za(\lambda z.zb(\lambda z.zc\,\text{nil})))$$

Die Liste beinhaltet drei Element(a,b,c). Das Ende wird mit nil gekennzeichnet. Es ist jedoch noch nicht klar welchem Term nil entspricht und wie das Ende der Liste erkannt werden kann. Es sind daher zwei Terme nil und null gesucht, die sich folgend verhalten:

$$\begin{aligned} \text{null}\,\text{nil} &= \text{true} \\ \text{null}(\text{pair}\,a\,\text{nil}) &= \text{false} \end{aligned}$$

Man definiert dazu:

nil	$\equiv$	$\lambda x.\,\text{true}$	Markiert das Ende der Liste.
null	$\equiv$	$\lambda x.x(\lambda yz.\,\text{false})$	Prüft ob das Ende der Liste erreicht ist.

Folgendes Beispiel soll die Sinnhaftigkeit der Definitionen von nil und null verdeutlichen:

$$\begin{aligned} \text{null}\,\text{nil} &\equiv (\lambda x.x(\lambda yz.\,\text{false}))(\lambda x.\,\text{true}) \rightarrow_\beta \\ &\rightarrow_\beta (\lambda x.\,\text{true})(\lambda yz.\,\text{false}) \rightarrow_\beta \text{true} \end{aligned}$$

$$\begin{aligned} \text{null}(\text{pair}\,a\,\text{nil}) &\equiv (\lambda x.x(\lambda yz.\,\text{false}))(\text{pair}\,a\,\text{nil}) \rightarrow_\beta \\ &\rightarrow_\beta (\text{pair}\,a\,\text{nil})(\lambda yz.\,\text{false}) \rightarrow_\beta (\lambda yz.\,\text{false})a\,\text{nil} \rightarrow^*_\beta \text{false} \end{aligned}$$

Die Bedeutung von fst für Listen ändert sich nicht, wobei man überlicherweise im Zusammenhang mit Listen von head spricht. Die Bedeutung des snd-Terms wird zu tail.

2.3 Church Numerale

Dieser Abschnitt soll die Natürlichen Zahlen einführen, die durch die Church Numerale repräsentiert werden. Die Natürlichen Zahlen können wie folgt veranschaulicht werden: Die Menge der Natürlichen Zahlen besteht aus der Null[3], dem Nachfolger der Null, dem Nachfolger des Nachfolgers der Null usw.

Definition Man definiert daher die Natürlichen Zahlen (die Church Numerale genannt werden) durch die oben beschriebene Vorschrift.

$0 \equiv \lambda sz.z$

$1 \equiv \lambda sz.sz$

[3] Soll die Null kein Element der Natürlichen Zahlen sein, so kann man auch mit Eins beginnen

$$2 \equiv \lambda sz.s(sz)$$

$$\vdots$$

$$\text{n} \equiv \lambda sz.s(s(\ldots(sz)\ldots) \equiv \lambda sz.s^n z$$

Durch Church Numerale ist es möglich Iterationen als Lambda-Term darzustellen. Wie man aus der Definiton erkennt gilt $nfa = \lambda sz.\underbrace{s(s(\ldots(s}_{n-mal}z)\ldots)fa$ was $\underbrace{f(f(\ldots(f}_{n-mal}(a)\ldots)$ entspricht.
Der Term a stellt also den Anfangszustand dar, während f die Zustandsänderungsfunktion ist. Durch das Church Numeral n wird bestimmt, wie oft iteriert werden soll.

Die iszero-Funktion Für iszero muss gelten $\text{iszero}\,0 = \text{true}$ und für alle Church Numerale n mit $n \neq 0$ $\text{iszero}\,n = \text{false}$.
Wie oben erwähnt, kann man nfa als Schleife deuten, wobei n ein Church Numeral ist. Im Falle der iszero-Funktion ist $f \equiv (\lambda y.\,\text{false})$ und $a \equiv \text{true}$. Wird der iszero-Funktion das Church Numeral 0 übergeben so wird die Schleife nicht durchlaufen, sondern sofort der Anfangszustand(a) zurückgegeben. Ist $n \neq 0$ wird die Schleife mindestens einmal durchlaufen. Die Zustandsänderungsfunktion macht aber nichts anderes als den vorhergehenden Wert wegzuwerfen und false zurückzugeben. Es ergibt sich $\text{iszero} \equiv \lambda n.n(\lambda x.\,\text{false})\,\text{true}$.

Die succ-Funktion Es sei n ein beliebiges Church Numeral. Für die succ-Funktion muss gelten $\text{succ}\,n = n + 1$. Der Term $\text{succ} \equiv \lambda nsz.s(nsz)$ erfüllt diese Bedingung:

$$\begin{array}{lcllcl}
\text{succ}\,n & \equiv & (\lambda nsz.s(nsz))n & \to_\beta & \lambda sz.s(nsz) \\
 & \equiv & \lambda sz.(s((\lambda sz.\underbrace{s(s(\ldots(s}_{n-mal}z)\ldots)))sz)) & \to_\beta^* & \lambda sz.s(\underbrace{s(s(\ldots(s}_{n-mal}z)\ldots))) \\
 & \equiv & \lambda sz.\underbrace{s(s(s\ldots(s}_{n+1-mal}z)\ldots)) & \equiv & n+1
\end{array}$$

Die Predecessor-Funktion Die Funktion pred ist eine Vorgängerfunktion. Für sie gilt $\text{pred}(n + 1) = n$ und $\text{pred}\,0 = 0$. Um sie zu realisieren kann man auf das Konzept der Paare zurückgreifen. Man betrachtet ein Paar (p_1,p_2) und eine zugehörige Funktion h, die das Paar (p_1,p_2) zu (p_2, p_2+1) ändert. Nach n-maliger Iteration erhält man an der ersten Position des Paares den Vorgänger von n. Für die Funktion h ergibt sich daher

$$h \equiv \lambda p.(\text{pair}(p\,\text{false})(\text{succ}(p\,\text{false})))$$

Aus dem Paar p wird das zweite Element entnommen und als erstes Element in ein neues Paar eingefügt. Das zweite Element des neuen Paares ist der Nachfolger des zweiten Elements des ersten Paares.
Es muss jetzt lediglich noch iteriert werden, was, wie oben gezeigt durch Church Numerale möglich ist: $pred' = \lambda n.nh(\text{pair}\,0\,0)$. Am Ende der Reduktion steht dann im ersten Element der Vorgänger von n. Dieser muss noch durch einen geeigneten Wahrheitswert herausgefiltert werden und es ergibt sich schließlich

$$\begin{array}{lcl}
\text{pred} & \equiv & (\lambda n.nh(\text{pair}\,0\,0))\,\text{true oder} \\
\text{pred} & \equiv & (\lambda n.n(\lambda p.(\text{pair}(p\,\text{false})(\text{succ}(p\,\text{false}))))(\text{pair}\,0\,0))\,\text{true}
\end{array}$$

Arithmetik mit Church Numeralen Mit Church Numeralen kann auch gerechnet werden. Dazu sind folgende Terme nützlich.

(i) $\text{add}\, nm = n + m$: $\text{add} \equiv \lambda nmsz.ns(msz)$ oder $\text{add} \equiv \lambda nm.n\, \text{succ}\, m$
Nachweis:
$\text{add}\, nm \rightarrow_\beta^* \lambda sz.ns(msz) \equiv \lambda sz.((\lambda sz.s^n z)s((\lambda sz.s^m z)sz)) \rightarrow_\beta^* \lambda sz.((\lambda z.s^n z)(s^m z)) \rightarrow_\beta$
$\rightarrow_\beta \lambda sz.s^n s^m z \equiv \lambda sz.s^{n+m} z \equiv n + m$

(ii) $\text{sub}\, nm = n - m$: $\text{sub} \equiv \lambda nm.m\, \text{pred}\, n$
Nachweis:
$\text{sub}\, nm \rightarrow_\beta^* m\, \text{pred}\, n \rightarrow_\beta^* \text{pred}(\text{pred}(\dots \text{pred}(n) \dots) \equiv \text{pred}^m\, n \equiv n - m$

(iii) $\text{mult}\, nm = n * m$: $\text{mult} \equiv \lambda nms.n(ms)$
Nachweis:
$\text{mult}\, nm \rightarrow_\beta^* \lambda s.((\lambda sz.s^n z)(\lambda z.s^m z)) \rightarrow_\beta \lambda s.\lambda z.((\lambda z.s^m z)^n z) \equiv$
$\equiv \lambda sz.((\lambda z.s^m z)((\lambda z.s^m z) \dots ((\lambda z.s^m z)z) \dots) \rightarrow_\beta^* \lambda sz.(s^m)^n z \equiv n * m$

(iv) $\exp nm = m^n$: $\text{expt} \equiv \lambda nm.nm$
Nachweis:
$\exp nm \equiv (\lambda sz.s^n z)(\lambda sz.s^m z) \rightarrow_\beta^* \lambda z.(\underbrace{((\lambda sz.s^m z)((\lambda sz.s^m z) \dots ((\lambda sz.s^m z)}_{n-mal}\, z) \dots) \rightarrow_\beta$

$\rightarrow_\beta \lambda z.((\lambda sz.s^m z)((\lambda sz.s^m z) \dots ((\lambda z_1.z^m z_1) \dots) \rightarrow_\beta$
$\rightarrow_\beta \lambda z.((\lambda sz.s^m z)((\lambda sz.s^m z) \dots ((\lambda z_2.((\lambda z_1.z^m z_1)^m z_2)) \dots) \rightarrow_\beta$
$\rightarrow_\beta \lambda z.((\lambda sz.s^m z)((\lambda sz.s^m z) \dots ((\lambda z_2.(\underbrace{z(z(\dots(z}_{m*m-mal}\, z_2) \dots) \rightarrow_\beta^*$

$\rightarrow_\beta^* \lambda z z_n.\, \underbrace{z(z(z(z(z(z(z(z(\dots(z}_{\underbrace{m * m * \dots * m}_{n-mal}-mal}\, z_n) \dots) \equiv m^n$

Beispiele

(i) $\text{add}\, 21 \equiv (\lambda nmsz.ns(msz))(\lambda sz.s(sz))(\lambda sz.sz) \rightarrow_\beta^* \lambda sz.((\lambda sz.s(sz))s(sz)) \rightarrow_\beta$
$\rightarrow_\beta \lambda sz.((\lambda z.s(sz))(sz)) \rightarrow_\beta \lambda sz.(s(s(sz))) \equiv 3$

(ii) $\text{sub}\, 32 \rightarrow_\beta^* 2\, \text{pred}\, 3 \overbrace{\rightarrow}^{\text{Schleife}} \text{pred}(\text{pred}\, 3) = 1$

(ii) $\text{mult}\, 22 \equiv (\lambda nms.n(ms))(\lambda sz.s(sz))(\lambda sz.s(sz)) \rightarrow_\beta^* \lambda s.((\lambda sz.s(sz))(\lambda z.s(sz))) \rightarrow_\beta$
$\rightarrow_\beta \lambda s.(\lambda z.((\lambda z.s(sz))((\lambda z.s(sz))z))) \rightarrow_\beta \lambda s.(\lambda z.((\lambda z.s(sz))(s(sz)))) \rightarrow_\beta$
$\rightarrow_\beta \lambda s.(\lambda z.(s(s(s(sz))))) \equiv \lambda sz.s(s(s(sz)))$

(iii) $\exp 22 \equiv (\lambda sz.s(sz))(\lambda sz.s(sz)) \rightarrow_\beta \lambda z.((\lambda sz.s(sz))((\lambda sz.s(sz))z)) \rightarrow_\alpha \rightarrow_\beta$
$\rightarrow_\alpha \rightarrow_\beta \lambda z.((\lambda sz.s(sz))(\lambda z_1.z(zz_1))) \rightarrow_\alpha \rightarrow_\beta \lambda z.(\lambda z_2.((\lambda z_1.z(zz_1))((\lambda z_1.z(zz_1))z_2))) \rightarrow_\beta$
$\rightarrow_\beta \lambda z.(\lambda z_2.((\lambda z_1.z(zz_1))(z(zz_2)))) \rightarrow_\beta \lambda z.(\lambda z_2.(z(z(z(zz_2))))) \equiv 4$

2.4 Rekursive Funktionen

2.4.1 Fixpunkte

Definition Ein Element x heißt Fixpunkt von f, falls gilt $f(x) = x$. Im Lambda-Kalkül wird ein Term t Fixpunkt von s genannt, falls gilt $st = t$.

Überraschend mag nun sein, dass es zu jedem Term s mindestens einen Fixpunkt gibt. In manchen Fällen sind die Fixpunkte sofort ersichtlich, wie für die Identität $\lambda x.x$, für die jeder Term ein Fixpunkt ist. Um jedoch einen Fixpunkt für jeden beliebigen Term zu konstruieren, benötigt man Fixpunktoperatoren.

Definition Ein Fixpunktoperator Y ist ein Term, der für jeden beliebigen Term s die Gleichung $Ys = s(Ys)$ erfüllt, d.h. Ys ist ein Fixpunkt von s.
Bis jetzt ist noch nicht bekannt, ob sich Fixpunktoperatoren im reinen Lambda-Kalkül definieren lassen. Dass es funktioniert zeigen, die Lösungen von A. Church und A. Turing.

Churchscher Fixpunktoperator Es sei $V_f \equiv \lambda x.f(xx)$ und $Y \equiv \lambda f.V_f V_f$, dann heißt Y Churchscher Fixpunktoperator. Im folgenden wird gezeigt, dass es sich tatsächlich um einen Fixpunktoperator handelt:

$$Yt \rightarrow_\beta V_t V_t \rightarrow_\beta t(V_t V_t) \leftarrow_\beta t(\lambda t.(V_t V_t))t \rightarrow_\beta t(Yt)$$

Turingsche Fixpunktoperator Es sei $A \equiv \lambda xf.f(xxf)$ und $\Theta \equiv AA$. Es handelt sich um einen Fixpunktoperator:

$$\Theta t \rightarrow_\beta AAt \rightarrow_\beta (\lambda f.f(AAf))t \rightarrow_\beta t(AAt) \equiv t(\Theta t)$$

2.4.2 Darstellung rekursiver Funktionen

Rekursive Funktionen rufen sich im Funktionsrumpf selbst wieder auf. Dazu muss jedoch die zu definierende Funktion im Funktionsrumpf bereits bekannt sein. Da im Lambda-Kalkül nur anonyme Funktionen existieren, ist dies nicht möglich. Es muss daher ein Weg gefunden werden, der es erlaubt, rekursive Funktion durch nicht-rekursive darzustellen.
Man betrachte dazu das Beispiel der folgenden Funktion zur Berechnung der n-ten Fakultät:

$$f = \lambda n.(\text{test}(\text{iszero}\, n)1(\text{mult}\, n(f(\text{pred}\, n))))$$

Die Variable f ist nicht definiert und daher führt die Funktion nicht zum gewünschten Ergebnis. Durch β-Expansion ergibt sich

$$f = \underbrace{(\lambda fn.(\text{test}(\text{iszero}\, n)1(\text{mult}\, n(f(\text{pred}\, n)))))}_{F} f$$

Man erhält die nicht-rekursive Funktion F. Um die rekursive Funktion f zu simulieren, muss man statt f immer wieder F selbst substituieren. Die Funktion F setzt sich nämlich dann, an der Stelle f immer wieder selbst fort, wobei genau das, das Ziel der rekursiven Funktion f war. Die Lösung bietet ein Fixpunktoperator (wie oben eingeführt). Da für einen Fixpunktoperator fix und einem beliebigen Term t gilt $\text{fix}\, t = t(\text{fix}\, t)$ erhält man

$\mathrm{fix}\,F = F(\mathrm{fix}\,F) = F(F(\mathrm{fix}\,F)) = F(F(F(\mathrm{fix}\,F))) = \ldots$. Die rekursive Funktion f entspricht daher $\mathrm{fix}\,F$.
Diese Vorgehensweise ist nicht auf die Funktion f des Beispiels beschränkt. Da nirgends explizit auf die Gesalt von f Bezug genommen wurde, lässt sich der Vorgang für jede rekursive Funktion durchführen.

2.4.3 Beispiel

Die Fibonacci-Folge:
Definition:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Lambda-Kalkül-ähnliche Definition:

```
fib = λn.
    (test (iszero n) 0 (
          test (iszero pred n) 1 (
             add (fib pred n) (fib (pred (pred n))))
          )
    )
```

Die Defintion des fib-Terms ist rekursiv. Der Term muss daher in eine nicht-rekursive Darstellung gebracht werden.

$$\mathrm{fib}' = \lambda f n.(\mathrm{test}(\mathrm{iszero}\,n)0(\mathrm{test}(\mathrm{iszero}\,\mathrm{pred}\,n)1(\mathrm{add}(f\,\mathrm{pred}\,n)(f(\mathrm{pred}(\mathrm{pred}\,n))))))$$

Mit dem Fixpunktoperator fix ergibt sich:

$$\mathrm{fib} = \mathrm{fix}\,\mathrm{fib}'$$

$$\begin{aligned} \text{fib 2} &= \text{fix fib' 2} \\ &\to_\beta^* \text{fib' (fix fib') 2} \\ &\to_\beta^* (\mathrm{test}(\mathrm{iszero}\,2)0(\mathrm{test}(\mathrm{iszero}\,1)1(\mathrm{add}((\mathrm{fix}\,\mathrm{fib}')1)(\mathrm{fix}\,\mathrm{fib}')0))) \\ &\to_\beta^* (\mathrm{add}(\mathrm{fib}'(\mathrm{fix}\,\mathrm{fib}')1)(\mathrm{fib}'(\mathrm{fix}\,\mathrm{fib}')0)) \\ &\to_\beta^* (\mathrm{add}\ \ (\mathrm{test}(\mathrm{iszero}\,1)0(\mathrm{test}(\mathrm{iszero}\,0)1(\mathrm{add}((\mathrm{fix}\,\mathrm{fib}')0)((\mathrm{fix}\,\mathrm{fib}')0)))) \\ &\qquad ((\mathrm{test}(\mathrm{iszero}\,0)0(\mathrm{test}(\mathrm{iszero}\,0)1(\mathrm{add}((\mathrm{fix}\,\mathrm{fib}')0)((\mathrm{fix}\,\mathrm{fib}')0)))))) \\ &\to_\beta^* (\mathrm{add}\,10) \\ &\to_\beta^* 1 \end{aligned}$$

2.4.4 Berechenbare Funktionen auf $\mathbb{N}$

Sowohl A. Church als auch A. Turing beschäftigen sich mit der Frage, welche Funktionen berechenbar sind. Es entstand die Church-Turing-These welche lautet:

> Alle vom Menschen berechenbaren Funktionen, können auch mit einer Turingmaschine berechnet werden.

Kann im Lambda-Kalkül eine Funktion berechnet werden, so nennt man sie λ-defnierbar.

Definition Eine Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt λ-definierbar, wenn es einen *geschlossenen reinen* λ-Term t gibt, für welchen gilt:

(i) $t\ \ m_1 \ldots m_n \to_\beta^* m$, falls $f(m_1 \ldots m_n) = m$,

(ii) $t\ m_1 \dots m_n$ besitzt keine β-Normalform, falls $f(m_1 \dots m_n)$ undefiniert ist.

Sowohl Turing als auch Church haben gezeigt, dass alle Turingmaschinen-berechenbaren Funktionen λ-definierbar sind, und umgekehrt. Es ist also zu Recht vom allmächtigen Lambda-Kalkül zu sprechen!

Literatur

[Han04] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists.* King's College Publ., 2004.

[KK06] Martin Kreuzer and Stefan Kühling. *Logik für Informatiker.* Pearson Studium, 2006.

[Met01] André Metzner. *Einführung in den λ-Kalkül.* 2001.

[Nan99] Sebastian Nanz. *Lambda-Kalkül als Programmiersprache.* 1999.

[Nip04] Tobias Nipkow. *Lambda-Kalkül.* 2004.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Wen05] Felix Weninger. *Der untypisierte Lambda-Kalkül.* 2005.