Ausarbeitung zum Vortag : Das Lambdakalkül

von Sebastian Sand

Gliederung

1) Einführung

Das Lambda-Kalkül ist die kleinste Programmiersprache der Welt. Es ist in der ersten Hälfte des letzten Jahrhunderts entstanden. Im Jahre 1900 stellte David Hilbert auf einem Mathematikerkongress seine berühmte Frage, ob es möglich sei mittels eines automatisierten Verfahrens welches die Axiome miteinander kombiniert, immer neue mathematische Wahrheiten zu erhalten, sodass man bei einer Laufzeit die lange genug gewählt sein muss, alle Gesetze der Mathematik erhält. Er stellte sich dabei die Mathematik wie einen Baum vor, dessen Stamm aus den Axiomen und aus Kombinationen von Axiomen besteht. Die Früchte des Baumes waren dabei die aus den Axiomen automatisiert entstandenen mathematischen Wahrheiten.

Die beiden Mathematiker Church und Turing beantworteten diese Frage mit nein.
Nach 1930 entwickelte Church das Lambdakalkül um die Frage, was ein automatisches Verfahren ist, zu beantworten. Mit Hilfe des Lambdakalküls zeigte er, dass eine Maschine (Computer) nie in der Lage sein würde aus den Axiomen alle Sätze der Mathematik herzuleiten, egal wie viel Leistung er hätte und wie viele Jahrhunderte man ihm Zeit gäbe.
Heute wird das Lambdakalkül in Sprachen wie LISP, Scheme, ML oder LEGO verwendet. Es dient auch als funktionale Programmiersprache. Bei der mathematisch exakten Beschreibung der denotationellen Semantik von funktionalen Programmiersprachen wird es ebenfalls eingesetzt. Man könnte also die Semantik des Lambdakalküls wiederum im Lambdakalkül erklären.
Das Lambdakalkül war nicht von Anfang an Perfekt. Es wurde im Laufe der Zeit immer weiterentwickelt. Dabei hatten viele Mathematiker darauf Einfluss. So machte beispielsweise erst Kleene die Subtraktion mittels des Lambdakalküls möglich, indem er die Vorläuferfunktion entwickelte.

Eine detaillierte Liste der historischen Entwicklungen findet sich unter :
http://www.tcs.informatik.uni-muenchen.de/~alti/lambda-vl/intro-slides-a.ps

2) Die Syntax des Lambdakalküls

Die Syntax des Lambdakalküls ist einfach und schnell erklärt, da sie nur aus vier Teilen besteht. Der <name> ist in der Regel ein beliebiger kleiner Buchstabe, kann aber grundsätzlich jedes Zeichen sein. Es gibt dabei lediglich wenige Ausnahmen, wie beispielsweise T, F, S oder P. Was diese Buchstaben im Einzelnen bedeuten wird im weiteren Verlauf der Ausarbeitung erklärt. Formal ausgedrückt :

<name> := a | b | c | d | …… | a1 | b1 | …… ▲ | ■ | ● | ☻

Beispiel : x

Die <expression> kann grundsätzlich alles sein. Alles bedeutet im Lambdakalkül entweder ein <name>, eine <function> oder eine <application>. Formal ausgedrückt:

<expression> := <name> | <function> | <application>

Beispiel : λ y . y

Die <function> besteht aus dem einzigen Symbol, dass im Lambdakalkül fest zugeordnet ist: Dem griechischen Buchstaben Lambda. Gefolgt wird das λ von einem <name> und einer <expression>. Der Punkt wird verwendet um <name> und <expression> zu trennen.
Formal ausgedrückt:

<function> := λ <name> . <expression>

Beispiel : (λ y . y) (λ y . y)

Zuletzt sei noch die <application> erklärt. Sie besteht aus zwei <expression>, welche durch einen Punkt voneinander getrennt werden. Da eine <expression>, wie bereits erläutert, alles sein kann, ist hier eine unendliche Verschachtelung möglich. Formal ausgedrückt:

<application> := <expression> . <expression>

Beispiel : (λ x . x (y))

3) Die Beta – Reduktion

Die einfache Funktion f (x) = x + 1 wird im Lambdakalkül wie folgt dargestellt:

f (x) = x + 1 → f = λ x . x + 1

Die Darstellung der Zahl 1 ist allerdings inkorrekt. Die korrekte Darstellung von Zahlen im Lambdakalkül wird im Punkt 7 – Arithmetik Teil 1 erklärt.
Bis auf das λ sind alle verwendeten Buchstaben willkürlich und können, innerhalb einer Expression, jederzeit unter Beachtung des Kontext ausgetauscht werden. Buchstaben sind im Lambdakalkül nur Platzhalter.

Beispiel : (λ x . x) y → (λ t . t) g → (λ ■ . ■) ▲

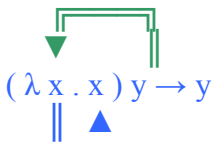
Das verwendete Beispiel wird als Identität von Lambda oder auch als Identitätsfunktion bezeichnet. Diese Funktion spielt bei der Darstellung von Zahlen eine große Rolle. Die Identitätsfunktion liefert als Ergebnis immer den eingesetzten Wert:

Beispiel : (λ x . x) y → [y / x] → y

(λ t . t) g → [g / t] → g

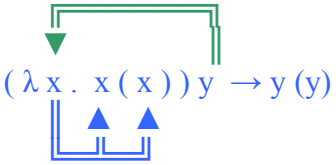
(λ ■ . ■) ▲ → [▲ / ■] → ▲

Das jeweilige Ergebnis erhält man durch Beta-Reduktion. Dabei wird der hinter der Funktion stehende Name in die Funktion eingesetzt. Dies erfolgt mittels folgender Systematik:



Das Einsetzen von einem Wert in die Funktion wird Formal durch [y / x] geschrieben. Gesprochen: „x wird durch y ersetzt“.

Ein weiteres Beispiel dazu:



Allgemein lässt sich das als (((E_1 E_2) E_3) E_4) schreiben. E_i steht dabei für jeweils eine Expression. Im folgenden Beispiel wird die Bedeutung klar:

(λ s . (λ z . s (s (z)))) f a

λ s ist die erste Expression E_1 in Form einer Funktion.

λ z . s (s (z)) ist die zweite Expression E_2 ebenfalls in Form einer Funktion.

f ist die dritte Expression E_3 in Form eines Name.

a ist die vierte Expression E_4 ebenfalls in Form eines Name.

Durch Beta-Reduktion werden von recht nach links die Namen in die Funktionen eingesetzt. Das Ergebnis sieht also folgendermaßen aus:

(λ s . (λ z . s (s (z)))) f a → (λ z . f (f (z))) a → f (f (a))

4.) Currying

Das Konzept des Currying wurde von Moses Schönfinkel entwickelt. Benannt wurde es aber nach dem Logiker Haskell B. Curry.
Unter Currying versteht man den Prozess, der aus einer Funktion mit n Parametern eine Funktion mit n-1 Parametern macht. Dieses Konzept wird beispielsweise in Scheme verwendet. Auch das Lambdakalkül arbeitet nach diesem Prinzip. Es wird also pro Schritt immer nur ein Argument der Funktion ausgewertet.

Beispiel: (λ x . (λ y . x y)) u v → (λ y . u y) v

Die Auswertung einer Funktion kann also als Ergebnis wieder eine Funktion haben.
Das Endergebnis der Funktion erhält man nach zwei Schritten: u v

Bei dem nachfolgenden weiteren Beispiel, wird wieder die im Lambdakalkül inkorrekte Zahlendarstellung verwendet, was an dieser Stelle allerdings egal ist, da es nur um das Prinzip der Funktionsauswertung geht.

f x y = x + y → f x y = λ x y . x + y → (λ 2) 3
1. Schritt → (λ y . 2 + y) 3
2. Schritt → (2 + 3)
3. Schritt → 5

Die Funktionsauswertung erfolgt also allgemein nach dem Schema:

X: int → (int → int)

5.) Reichweite (scope) von Variablen

Im Lambdakalkül unterscheidet man zwischen freien und gebundenen Variablen. Jede Variable, die innerhalb einer Expression durch ein λ eingeführt wurde, ist gebunden. Daraus folgt, dass Variablen die nicht durch ein λ eingeführt worden sind, freie Variablen sind.

Beispiel: (λ x . (λ y . x y)) y

Die Variablen x und y sind hier beide innerhalb der Funktion gebunden. y ist aber auch eine freie Variable in der Expression mit der Form Name. Wie bereits an vorangegangener Stelle erläutert, sind Variablen im Lambdakalkül nur Platzhalter und dürfen unter Beachtung des Kontextes beliebig ausgetauscht werden.

(λ x . (λ y . x y)) y = (λ ▲ . (λ ■ . ▲ ■) y

Diesen Vorgang des Variablen Austauschens nennt man Alpha–Konversion. Nach durchgeführter Alpha-Konversion, kann man wie gehabt mit der Beta-Reduktion fortfahren. Bei der Alpha-Konversion werden komplett neue Variablennamen generiert. Formal geschrieben sieht das wie folgt aus:

(λ x . (λ y . x y)) y → [s / x] [t / y] → (λ s . (λ t . s t) y

Jetzt bestehen im Beispiel keinerlei Konflikte mehr zwischen gebundenen und freien Variablen. Eine Beta-Reduktion ist also möglich.

6.) Freie und gebundene Variablen

Es kann sowohl in Applikationen, als auch in Funktionen, freie und gebundene Variablen geben. Im nachfolgenden Beispiel ist x gebunden und y frei.

Beispiel : (λ x . x y)

In derart trivialen Funktionen ist es nicht schwer herauszufinden ob die jeweiligen Variablen frei oder gebunden sind. Um dies bei komplexeren Funktionen oder Applikationen zu ermitteln gibt es eine Reihe von formellen Regeln. Zuerst die Regeln für die freien Variablen:

1. <name> ist frei in <name>

Beispiel : y [y ist frei]

2. <name> ist frei in λ <name$_1$> . <expression> falls <name> ≠ <name$_1$> und <name> frei in <expression>

   Beispiel: (λ x . x y) [y ist frei]

3. <name> ist frei in E_1 E_2 falls <name> in E_1 frei ist oder <name> in E_2 frei ist

   Beispiel: (λ x . x) (λ z . y z) [y ist frei]

Auch für die gebundenen Variablen gibt es entsprechende Regeln:

1. <name> ist gebunden in λ <name$_1$> . <expression> falls <name> = <name$_1$> oder <name> in <expression> gebunden ist.

   Beispiel: (λ x . x) [x ist gebunden]

2. <name> ist gebunden in E_1 E_2 falls <name> gebunden ist in E_1 oder <name> gebunden ist in E_2.

   Beispiel: (λ y . y) (λ x . x) [x ist gebunden]

Einer der katastrophalsten Fehler bei Berechnungen mit dem Lambdakalkül ist das einsetzen von freien Variablen, an Stellen an denen sie gebunden sind. Deshalb: Setze nie freie Variablen in einen Ausdruck ein, in dem die Variable gebunden ist.

7.) Arithmetik Teil 1

Um mit dem Lambdakalkül Mathematik betreiben zu können, muss zuerst die korrekte Schreibweise für Zahlen eingeführt werden. Zahlen werden im Lambdakalkül mittels der Identitätsfunktion [λ x . x] dargestellt. Die Anzahl der ineinander verschachtelten Identitätsfunktionen bestimmt den Wert der Zahl.

$0 \equiv (\lambda s . (\lambda z . z))$
$1 \equiv (\lambda s . (\lambda z . s (z)))$
$2 \equiv (\lambda s . (\lambda z . s (s (z))))$
$3 \equiv (\lambda s . (\lambda z . s (s (z))))$ usw.....

Dies wird auch als iterative Funktionsapplikation bezeichnet. Dazu ein Beispiel:

$f\ a$
$f^0 a \rightarrow a$
$f^1 a \rightarrow f (a)$
$f^2 a \rightarrow f (f (a))$
$f^3 a \rightarrow f (f (f (a)))$

$2\ f\ a \equiv (\lambda s (\lambda z . s (s (z))))\ f\ a$

$\rightarrow (\lambda z . f(f(z))) a$
$\rightarrow f(f(a))$

Mit diesem Wissen kann man nun die Nachfolgerfunktion einführen. Die Nachfolgerfunktion wird mit S abgekürzt.
$S \equiv (\lambda w . (\lambda y . (\lambda x . y (w y x))))$
$\equiv (\lambda w y x . (y (w y x))$

$S 2 \equiv (\lambda w y x . y (w y x)) 2$
$\rightarrow (\lambda y x . y \underline{(2 y x)})$ [Church – Rosser – Satz]
$\rightarrow (\lambda y x . y (y (y (x)))) \rightarrow 3$

Das Ergebnis ist die Zahl 3. Dies ergibt sich aus der Definition der Zahl:
$3 \equiv (\lambda s . (\lambda z . s (s (s (z))))) = (\lambda s z . s (s (s (z))))$

Die Nachfolgerfunktion von 3, also S3 ist somit 4 und die Nachfolgerfunktion von 4, also S4 ist somit 5.
Mit diesen Kenntnissen sind erste Berechnungen im Lambdakalkül möglich. Die einfachste Rechnung im Lambdakalkül ist die Addition.
Will man zwei Zahlen x und y addieren, so wertet man x mal die Nachfolgerfunktion von y aus: x S y. Dazu ein Beispiel:

$2 + 3 = 2 S 3 \rightarrow S (S (3)) \rightarrow S (4) \rightarrow 5$

Die allgemeine Additionsfunktion lautet: $(\lambda x y . x S y)$. Angewandt auf das letzte Beispiel, ergibt sich folgende Rechnung:

$(\lambda x y . x S y) 2 3 \rightarrow (\lambda y . 2 S y) 3 \rightarrow 2 S 3 \rightarrow 5$

Weit weniger einfach und übersichtlich als die Addition ist die Multiplikation im Lambdakalkül. Die allgemeine Multiplikationsfunktion lautet:

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

Dazu ein Beispiel: $2 \cdot 2$

$(\lambda x y z . x (y z)) 2 2$
$\rightarrow$ [2 wird für x eingesetzt]
$\rightarrow (\lambda y z . 2 (y z)) 2$
$\rightarrow$ [2 wird für y eingesetzt]
$\rightarrow (\lambda z . (2 (2 z))$
$\rightarrow$ [Für 2 wird die Entsprechung im Lambdakalkül eingesetzt]
$\rightarrow (\lambda z . (\lambda s z . s (s (z))) ((\lambda s z . s (s (z))) z))$
$\rightarrow$ [Da z gebunden ist, kann es nicht durch z ersetzt werden. Wir führen also eine α-Konversion durch.]
$\rightarrow (\lambda z . (\lambda s z . s (s (z))) ((\lambda s t . s (s (t))) z))$
$\rightarrow$ [s wird durch z ersetzt]
$\rightarrow (\lambda z . (\lambda s z . s (s (z))) ((\lambda t . z (z (t)))))$
$\rightarrow$ [Da z im ersten Teil der Applikation gebunden ist, kann man nicht den zweiten Teil für s einsetzen. Es käme sonst zu Konflikten mit z. Wir führen also eine α-Konversion durch.]
$\rightarrow (\lambda z . (\lambda s a . s (s (a))) ((\lambda t . z (z (t)))))$

→ [(λ t . z (z (t))) wird für s eingesetzt.]
→ (λ z . (λ a . (λ t . z (z (t))) (λ t . z (z (t))) (a)))
→ [t wird durch a ersetzt]
→ (λ z . (λ a . (λ t . z (z (t)))) (z (z (a)))
→ [t wird durch (z (z (a))) ersetzt]
→ (λ z . (λ a. z (z (z (z (a))))))
→ 4

8.) Booleans

Als Erstes definieren wir die beiden Werte True und False im Lambdakalkül. Im Lambdakalkül sind True und False Entscheidungsträger die immer das erste nachfolgende Element bzw. das zweite nachfolgende Element zurück liefern.

T ≡ (λ x y . y)

F ≡ (λ x y . x)

IF ist eine Abwandlung der Funktionen für True bzw. False. Die allgemeine Definition von IF lautet:

IF ≡ (λ b x y . b x y)

b ist hierbei entweder True oder False. Je nachdem entscheidet es sich entweder für x oder y. Dazu ein Beispiel:

IF (λ b x y . b x y) T 3 4 → T 3 4 → 3

Als nächstes wird AND allgemein definiert :

/\ ≡ (λ x y . x y (λ u v . v)) → [(λ u v . v) = F] → (λ x y . x y F)

Zum bessern Verständnis sehen wir uns die folgenden zwei Beispiele an :

Beispiel 1:

/\ T T ≡ (λ x y . x y F) T T
→ (λ y . T y F) T
→ T T F
→ T [denn T liefert immer x als Ergebnis]

Beispiel 2:

/\ F F ≡ (λ x y . x y F) F F
→ (λ y . F y F) F
→ F F F
→ F

Der nächste Boolean dem wir uns widmen ist OR. Die allgemeine Definition von OR ist

$V \equiv (\lambda x y . x (\lambda x y . x) y)$
$\equiv (\lambda x y . x T y)$

Zur Verdeutlichung folgen wieder zwei Beispiele :
Beispiel 1:

$V T F \equiv (\lambda x y . x (\lambda x y . x) y) T F$
$\rightarrow (\lambda y . T T y) F$
$\rightarrow T T F$
$\rightarrow T$

Beispiel 2 :

$V F F \equiv (\lambda x y . x (\lambda x y . x) y) F F$
$\rightarrow (\lambda y . F T y) F$
$\rightarrow F T F$
$\rightarrow F$ [denn F liefert immer y als Ergebnis]

Als letzter Boolean schauen wir uns NOT an. Die allgemeine Definition von NOT ist

$\neg \equiv (\lambda x . x F T)$

Zur Veranschaulichung wieder zwei Beispiele:

Beispiel 1 :

$\neg T \equiv (\lambda x . x F T) T$
$\rightarrow T F T$
$\rightarrow F$

Beispiel 2:

$\neg F \equiv (\lambda x . x F T) F$
$\rightarrow F F T$
$\rightarrow T$

NOT liefert also immer das Gegenteil des eingesetzten Wertes.

Mit Hilfe der Booleans ist nun ein so genannter „Conditional Test“ möglich, welcher überprüft, ob eine Zahl 0 ist oder nicht. Die allgemeine Form des „ Conditional Test“, kurz Z, ist die Folgende:

$Z \equiv (\lambda x . x F \neg F)$

Es ist wichtig hier an die Regel zu erinnern, dass immer von links nach rechts assoziiert wird:

$Z \equiv (\lambda x . (((x F) \neg) F))$

Wie wir wissen ergibt sich im Lambdakalkül die Zahl n, indem man n-Mal die Identitätsfunktion auf ein Argument a anwendet:

n f a → f (f (f (….. f (a)) ….)
Die Zahl 0 ist also 0 f a → a

Mittels Z können wir nun überprüfen ob n = 0 ist oder nicht.

Z 0 ≡ (λ x . x F ¬ F) 0
[0 wird für x eingesetzt.]
→ 0 F ¬ F
[F wird jetzt 0-Mal auf ¬ angewendet. ¬ bleibt also unberührt.]
→ ¬ F
→ T

Wir haben also gezeigt, dass die Zahl 0 = 0 ist.

Um nun aber zu überprüfen ob eine Zahl n = 0 ist, werten wir Z n aus:

Z n ≡ (λ x . x F ¬ F) n
→ n F ¬ F
[Hier sollte man sich in Erinnerung rufen, dass F aus (x , y) immer y auswählt. Setzt man aber nur ein x in F ein ergibt sich : F a ≡ (λ x y . y) a → (λ y . y). Also die Identitätsfunktion. Wendet man das F also auf ¬ an, ergibt sich die Identitätsfunktion. Dies tut man n-Mal. Da F I = I ist, ergibt sich für n-Mal F auf ¬ angewendet immer I.]
→ I F
[Die Identität von F = F.]
→ F

Zu diesem Ergebnis kommt man nur, wenn F mindestens einmal auf ¬ angewandt wurde, n also ≠ 0 ist.

9.) Rekursion

Die Allgemeine Definition der rekursiven Funktion im Lambdakalkül ist wie folgt:

Y ≡ λ y . (λ x . y (x x)) (λ x . y (x x)))

Y ist der Rekursionsopperator.

Nachfolgend soll R die Funktion sein die wir später rekursiv aufrufen wollen. Setzt man ganz allgemein R in den Rekursionsopperator, so ergibt sich:

Y R = λ y . (λ x . y (x x)) (λ x . y (x x)) R
→ [R wird für y eingesetzt.]
→ (λ x . R (x x)) (λ x . R (x x))
→ [Das zweite (λ x . R (x x) wird für x eingesetzt.]
→ R ((λ x . R (x x)) (λ x . R (x x)))
→ R (Y R)

→ [Y erstellt also eine Kopie von R. Das ergibt R (Y R).]

Die Rekursion von R erzeugt die Anwendung von R auf YR => R (YR).

Die sicherlich bisher komplizierteste Anwendung des Lambdakalküls sollte anhand des folgenden Beispiels klar werden.

$0 + 1 + 2 + 3 + 4 + \ldots + n = S_n$

$S_n = n + S_{n-1}$
$S_0 = 0$

$R \equiv \lambda\ r\ n\ .\ Z\ n\ 0\ (\ n\ S\ (\ r\ (\ P\ n\ )\ )\ )\ )$ [Definition von S_n]

Das erste Argument von λ „r“ steht für S_n, das zweite Argument „n“ steht für eine Zahl.
Z n 0 überprüft ob n = 0 ist und liefert, falls dies der Fall ist 0 als Ergebnis.
(n S (r (P n))) steht für den fall n ≠ 0. Dann wird n-Mal die Nachfolgerfunktion auf r (P n) angewendet.

Y R 1 ≡ R (Y R) 1
→ (λ r n . Z n 0 (n S (r (P n)))) (Y R) 1
→ [(Y R) für r und 1 für n }
→ Z 1 0 (1 S ((Y R) (P 1)))
→ [Da Z 1 nicht 0 ist, wird nicht das erste Argument 0, sonder das zweite Argument ausgewählt. Das ganze reduziert sich also auf:]
→ 1 S ((Y R) (P 1))
→ [(Y R) ist ein rekursiver Aufruf, der R (Y R) ergibt.]
→ 1 S (R (Y R) (P 1))
→ [Die Rekursion wird einmal durchlaufen, weswegen man für R wieder S_n einsetzt.]
→ 1 S ((λ r n . Z n 0 (n S (r (P n)))) (Y R) (P 1))
→ [(Y R) wird für r und (P 1) für n eingesetzt]
→ 1 S (Z (P 1) 0 ((P 1) S ((Y R) (P (P (1))))))
→ [Da (P 1) = 0 ist, wählt Z 0 das erste Argument 0 aus. Es ergibt sich also:]
→ 1 S 0
→ $1 = S_1$

Wie schnell die Rekursion im Lambdakalkül unübersichtlich wird, sieht man wenn man statt der 1, wie im letzten Beispiel, die 3 einsetzt.

Y R 3 fl (Y R) 3
→ (λ r n . Z n 0 (n S (r (P n)))) (Y R) 3
→ [R soll 3 mal durchlaufen werden. Müsste als Ergebnis also 3 + 2 + 1 = 6 liefern]
→ [für r wird die rekursive Anwendung (Y R) eingesetzt]
→ λ n . Z n 0 (n S (Y R (P n)))) 3
→ [für n wird 3 eingesetzt]

→ Z 3 0 (3 S (Y R (P 3)))
→ [Da Z 3 nicht 0 ist, wird nicht das erste Argument 0, sonder das zweite Argument ausgewählt. Das ganze reduziert sich also auf:]
→ 3 S (Y R (P 3))
→ [3 S bedeutet 3-Mal die Nachfolgerfunktion von (Y R (P 3)), also 3 +. Das P 3
→ ergibt 2 (Vorläufer von 3 ist 2.). Es bleibt also (Y R (2)) übrig.]
→ [für YR wird reduziert auf R (Y R).]
→ 3 S (R (Y R) (P 3))
→ 3 S (P 3) S R (Y R) (P (P 3))
[an dieser Stelle wird die Rekursion noch 3 mal durchlaufen und führt dann zu dem Ergebnis:]
→ 3 S 2 S 1 S 0
→ 3 + 2 + 1
→ 6

10.) Arithmetik Teil 2

Wie bereits eingangs erwähnt, ist das heutige Lambdakalkül um viele Funktionen erweitert. So geht die Vorläuferfunktion, oder auch Predecessor Function, auf Kleene zurück. Die Vorläuferfunktion wird im Lambdakalkül mit P abgekürzt.
Um mit der Vorläuferfunktion rechnen zu können, ist es unerlässlich Paare einzuführen. Paare werden im Lambdakalkül als (λ z . z a b) geschrieben.

Beispiele für Paare : (0 , 0)
(1 , 0)
(2 , 1)
(3 , 2)
(4 , 3)
allgemein: $p = (p_1 , p_2)$

In Paaren ist neben der ersten Zahl p auch immer eine zweite Zahl der Art p-1 enthalten. Die Null ist hier ein Spezialfall, da negative Zahlen nicht dargestellt werden können.
Wendet man T bzw. F auf ein Paar an so erhält man immer p_1 oder p_2.

(λ z . z a b) T → T a b → a

(λ z . z a b) F → F a b → b

Um die Vorläuferfunktion herleiten zu können benötigt man die Hilfsfunktion

$\Phi \equiv \lambda p . (S (p_1)) (p_1))$

In diese Hilfsfunktion wird nun das Paar (1 , 0) eingesetzt.

Φ (λ t . t 1 0) ≡ (λ p . (S (p T)) (p T)) (λ t . t 1 0)
→ [(λ t . t 1 0) wird für p eingesetzt]
→ (S (λ t . t 1 0) T)) (λ t . t 1 0) T))
→ [T wird für t eingesetzt.]
→ S (1) (1)

→ [Die Nachfolgerfunktion S (1) ist 2]
→ (2 , 1)

Daraus kann man die Vorläuferfunktion folgern:

P ≡ (λ n . n Φ (λ z . z 0 0) F)

In dieser Funktion bildet die Hilfsfunktion aus einer Zahl p das Paar mit p. In dem entsprechenden Paar von p wird auch immer die Zahl p-1 dargestellt. (λ n . n) am Anfang von P sorgt dafür, dass die Hilfsfunktion p-mal durchlaufen wird um das richtig Paar zu bilden. Der Selektor F wählt dann aus dem gebildeten Paar p-1 aus und man erhält den Vorläufer von p. Hierzu ein Beispiel : Wir suchen den Vorläufer von 1:

P 1 ≡ λ n . n Φ (λ z .z 0 0) F) 1
→ [1 wird für n eingesetzt]
→ 1 Φ (λ z . z 0 0) F
→ [1 fällt weg, da Φ nur einmal durchlaufen werden muss.]
→ Φ (λ z . z 0 0) F
→ [Φ wird einmal durchlaufen und wirft das Paar (1 , 0) aus.]
→ (λ z . z 1 0) F
→ F 1 0
→ 0

Neben der Addition und Multiplikation ist nun auch die Subtraktion im Lambdakalkül möglich. Eine vereinfachte und abgekürzte Rechnung mit P sieht wie folgt aus.

5 – 3 = 3 P 5 = P (P (P (5))) = P (P (4)) = P (3) = 2

11.) Quellen

- Thorsten Altenkirch: „λ-Kalkül und Typen – 0. Einführung und Geschichte des λ-Kalküls“
- Fabian Nilius: „Das gefürchtete λ-Kalkül – Eine Einführung“
- Raúl Rojas: „A Tutorial Introduction to the Lambda Calculus“
- Friedman, Wand, Haynes: „Essentials of Programming Languages – Kapitel 4: Reduction Rules and Imperative Programming”
- Matthias Felleisen: „A Lecture on the Why of Y”
- Alexander Gloye: http://page.inf.fu-berlin.de/~gloye/lambda38/lambda38.cgi