Der untypisierte Lambda-Kalkül

Kern einer minimalen Programmiersprache

Sabine Rieder

Inhaltsverzeichnis

1 Einleitung

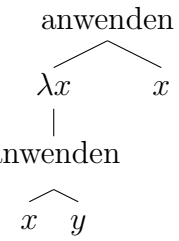
Der Lambda-Kalkül wurde in den Dreißigerjahren des letzten Jahrhunderts von Alonzo Church und Stephen Kleene entwickelt. Grundgedanke ist, Funktionen nicht, wie in der Mathematik üblich, als Abbildung von einer Menge in eine andere zu betrachten, sondern sie als Rechenvorschrift aufzufassen. Bei der Funktion $f : D \to B$ stehen also nicht die Mengen und die Beziehung eines Elementes des Definitionsbereiches zu einem des Bildbereiches im Vordergrund. Viel mehr wird auf die genaue Berechnung Wert gelegt. Das heißt, zu jedem Argument wird ein Funktionswert berechnet. Mengen als Bild- und Definitionsbereich haben dadurch im untypisierten Lambda-Kalkül keine Bedeutung [3, S. 23 f.].
Aufgrund dieser Auffassung von Funktionen kann man den Lambda-Kalkül dazu nutzen, eine Programmiersprache zu entwerfen. Genauer gesagt, ist er sogar der Kern der funktionalen Sprachen. Im weiteren Verlauf soll sich erst näher mit dem Kalkül befasst werden, im Anschluss werden die Grundlagen einer Programmiersprache entwickelt. Dabei orientiert sich die vorliegende Arbeit an [4, S. 51–73]. Allerdings wird hier keine formale Definition des Lambda-Kalküls erfolgen.

2 Grundlagen des Lambda-Kalküls

Im Lambda-Kalkül werden Rechenoperationen beschrieben, die auf ein beliebiges Eingangsparameter, auch einen anderen Term, angewendet werden können und, sofern ein Ergebnis existiert, dieses zurückliefern. Im folgenden Abschnitt wird zuerst die Syntax behandelt. Danach soll untersucht werden, wie die Funktionen ausgeführt werden und im Anschluss soll noch Substitution betrachtet werden.

2.1 Syntax

Bei der Syntax des Lambda-Kalküls treten drei verschiedene Fälle auf [3, S. 25]. Zuerst einmal sind Variable und Konstanten, wie zum Beispiel x, atomare Lambda-Terme. Außerdem ist die Anwendung, auch Applikation genannt, eines Terms auf einen anderen wieder ein Lambda-Term. So auch $(M\ N)$. Hier wird der Term M auf den Term N angewendet. Die letzte Art eine Terms ist noch die Abstraktion. Hier wird beispielsweise bei $(\lambda x.M)\, N$ jedes vorkommen der Variable x im Term M durch den Term N ersetzt. $(\lambda x.M)$ ist dabei die Abstraktion. Beispiele für Lambda-Terme sind $(\lambda x.x)M = M$, was die Identität darstellt, oder auch $(\lambda x.y)M = y$, wobei y eine Variable ist.
Für den Umgang mit Variablen benötigt man noch die Begriffe „gebunden“ und „frei“. Eine Variable, die unterhalb von einem Lambda im Syntaxbaum steht, ist gebunden. Alle Variablen, die nicht unterhalb eines Lambdas stehen, sind frei [4, S. 54 f.]. Die Menge der freien Variablen des Terms M bezeichnet man mit $FV(M)$. Eine Variable kann gleichzeitig gebunden und frei in einem Term vorkommen. Im Term $(\lambda x.x\, y)x$ sind beispielsweise die Variablen x und y frei, das x kommt aber auch gebunden vor, wie deutlich wird, wenn man den Syntaxbaum betrachtet:



Wichtig ist weiterhin, dass nach einem Lambda immer nur eine Variable stehen darf [4, S. 58]. Benötigt eine Berechnung mehrere Eingangsparameter, so werden diese geschachtelt aufgeschrieben, also $\lambda x.\lambda y.\lambda z.\ldots$.

2.2 Reduktion

Bei der Reduktion wird prinzipiell das Programm, das ein Lambda-Term darstellt, ausgeführt. Es gibt zwei wichtige Arten. Die α-Konversion ist die Umbenennung von Variablen [1, S. 100 f.], also zum Beispiel $\lambda x.x \underset{\alpha}{=} \lambda y.y$. Diese verändert den Sinn nicht und wird später bei der Substitution noch einmal wichtig. Bei der β-Reduktion werden die Terme anstelle der hinter dem Lambda angegebenen Variablen eingesetzt [1, S. 100]. Sie ist die eigentliche Programmausführung. Ein Beispiel dafür ist $(\lambda x.x\,y\,x)M \;\rightarrow\; M\,y\,M$. Üblicherweise wird die β-Reduktion durch einen Pfeil deutlich gemacht.

2.3 Substitution

Bei der Substitution sollen systematisch Variablen umbenannt und durch andere Terme ersetzt werden. Dafür müssen natürlich bestimmte Regeln aufgestellt werden.

2.3.1 Naiver Ansatz

Der naive Ansatz stellt relativ leichte Regeln zur Verfügung [4, S. 69 ff.]:

$$[x \to s]\,x = s \tag{1}$$

$$[x \to s]\,y = y \qquad \text{wenn } x \neq y \tag{2}$$

$$[x \to s]\,(\lambda y.M) = \lambda y.[x \to s]\,M \tag{3}$$

$$[x \to s]\,(M\ N) = ([x \to s]\,M)([x \to s]\,N) \tag{4}$$

Im ersten Fall wird eine ungebundene Variable umbenannt, im zweiten zeigt die Substitution keine Wirkung, da x nicht im Term vorkommt. Im dritten Fall wird die Substitution hinter λy geschoben und im letzten Fall werden die beiden Terme getrennt.
Allerdings treten jetzt einige Probleme auf. Im nachfolgenden Beispiel wird (3) angewendet, wie es der Definition entspricht:

$$[z \to y]\,\lambda z.z \;=\; \lambda z.[z \to y]\,z \;=\; \lambda z.y$$

Das Ergebnis ist aber nicht gleich zum Ausgangsterm, sondern die Bedeutung hat sich massiv verändert. Gleiches geschieht hier:

$$[x \to y]\, \lambda y.x \;=\; \lambda y.[x \to y]\, x \;=\; \lambda y.y$$

An dieser Stelle wird durch (3) das eigentlich freie z gebunden.

2.3.2 Verbesserter Ansatz

Beim verbesserten Ansatz wird folglich (3) verändert [4, S. 71]:

$$[x \to s]\, x = s \tag{1}$$
$$[x \to s]\, y = y \qquad \text{wenn } x \neq y \tag{2}$$
$$[x \to s]\, (\lambda y.M) = \lambda y.[x \to s]\, M \qquad x \neq y \text{ und } y \notin FV(s) \tag{3}$$
$$[x \to s]\, (M\, N) = ([x \to s]\, M)([x \to s]\, N) \tag{4}$$

Die erste Bedingung $x \neq y$ verhindert ein Umbenennen gebundener Variablen. Die zweite wirkt einem Binden von freien Variablen in s entgegen. Mit Hilfe der α-Reduktion kann in diesem Fall dennoch das s eingesetzt werden:

$$[x \to\ y]\lambda y.x\, y \underset{\alpha}{=} [x \to y]\lambda z.x\, z = \lambda z.y\, z$$

Hier wurde die gebundene Variable y mit z bezeichnet, sodass nun keine Überschneidungen mehr auftreten.

3 Lambda-Kalkül als Programmiersprache

Wie bereits angekündigt, soll jetzt eine kleine Programmiersprache entwickelt werden. Dafür sind Wahrheitswerte und auch Zahlen notwendig. Diese sollen nun im Lambda-Kalkül codiert werden.

3.1 Boolesche Logik

Für die Boolesche Logik sind natürlich *true* und *false* sowie logische Verknüpfungen nötig. Sie sollen nun genauer betrachtet werden.

3.1.1 Wahrheitswerte und bedingte Ausführung

Die Wahrheitswerte können auf verschiedene Weisen im Lambda-Kalkül dargestellt werden [1, S. 107]. Hier wird die folgende Codierung verwendet [4, S. 58 f.]:

$$\text{true} = \lambda t.\lambda f.t$$
$$\text{false} = \lambda t.\lambda f.f$$

Wieso diese Codierung gut geeignet ist, wird deutlich, wenn man bedingte Ausführung, also Konstrukte nach „if-then-else“ Schema, betrachtet. Der Test wird dabei wie folgt dargestellt [4, S. 59]:

$$\text{test} = \lambda l.\ \lambda m.\ \lambda n.\ l\, m\, n$$

Diese Formel führt je nachdem, ob der Wert, der für l eingesetzt wird, wahr oder falsch ist, m oder n aus. Setzt man für b *true* ein, wird das Konstrukt wie folgt ausgewertet:

$$\begin{aligned}
(\lambda l.\ \lambda m.\ \lambda n. l\, m\, n\,) b\, v\, w &= \\
(\lambda l. \lambda m. \lambda n. l\, m\, n)\ \text{true}\ v\, w &\rightarrow \\
(\lambda m. \lambda n. \text{true}\ m\, n) v\, w &\rightarrow \\
(\lambda n. \text{true}\ v\, n\,) w &\rightarrow \\
(\text{true}\ v\, n) &= \\
(\lambda t. \lambda f. t) v\, w &\rightarrow \\
(\lambda f. v) w &\rightarrow \\
v &
\end{aligned}$$

Dabei kennzeichnet das Gleichheitszeichen das Einsetzen eines Terms und der Pfeil die β-Reduktion.

3.1.2 logische Verknüpfungen

In diesem Kapitel sollen die Verknüpfungen *and* und *not* behandelt werden. *and* lässt sich wie folgt darstellen [3, S. 39]:

$$\text{and}\ = \lambda b. \lambda c. b\, c\ \text{false}$$

Führt man *and true true* aus, verhält sich der Term folgendermaßen:

$$\begin{aligned}
\text{and true true}\ &= \\
(\lambda b. \lambda c. b\, c\ \text{false})\text{true true} &\rightarrow \\
(\lambda c. \text{true}\ c\ \text{false})\text{true} &\rightarrow \\
\text{true true false} &= \\
(\lambda t. \lambda f. t)\text{true false} &\rightarrow \\
\text{true} &
\end{aligned}$$

Es wird also abhängig vom ersten Wert nach dem *and* gewählt. Handelt es sich dabei um *true* wird der zweite Wert zurückgegeben. Ist es ein *false* wertet der Term zu *false* aus, da das *false*, das ganz am Ende des *and*-Terms steht, zurückgeliefert wird.
Ähnlich funktioniert es bei *not* [3, S. 39]:

$$\text{not}\ = \lambda b. b\ \text{false true}$$

Auch hierfür wieder ein Beispiel:

$$\begin{aligned}
\text{not false} &= \\
(\lambda b.b\ \text{false true})\text{false} &\rightarrow \\
\text{false false true} &= \\
(\lambda t.\lambda f.f)\text{false true} &\rightarrow \\
\text{true}&
\end{aligned}$$

Wie man sieht, wird auch hier wieder ausgenutzt, dass *true* den ersten und *false* den zweiten Term nach sich selbst zurückliefert. Die übrigen logischen Verknüpfungen wie „or“ können nun vom interessierten Leser selbst gebildet oder in geeigneter Literatur nachgeschlagen werden.

3.2 Paare

Es werden nun Paare von Werten behandelt [4, S. 60]. Dabei soll jeweils entweder der erste oder der zweite Wert zurückgeliefert werden. Paare sind beispielsweise wichtig für das Dekrementieren von Zahlen [4, S. 62] und werden deshalb vorgestellt, auch wenn diese Operation hier nicht mehr behandelt wird.

$$\begin{aligned}
\text{pair} &= \lambda f.\lambda s.\lambda b.\, b\ f\ s && \\
\text{first} &= \lambda p.p\ \text{true} && \text{wobei p ein pair ist} \\
\text{second} &= \lambda p.p\ \text{false} && \text{wobei p ein pair ist}
\end{aligned}$$

Beispielsweise soll nun der erste Wert eines Paares zurückgegeben werden:

$$\begin{aligned}
\text{first (pair } v\, w) &= \\
(\lambda p.p\ \text{true})(\text{pair } v\, w) &\rightarrow \\
\text{pair } v\, w\ \text{true} &= \\
(\lambda f.\lambda s.\lambda b.b\ f\ s)v\, w\ \text{true} &\rightarrow \\
(\lambda s.\lambda b.b\, v\, s)w\ \text{true} &\rightarrow \\
(\lambda b.b\, v\, w)\text{true} &\rightarrow \\
\text{true}\ v\, w &= \\
(\lambda t.\lambda f.t)v\, w &\rightarrow \\
v&
\end{aligned}$$

Man stellt fest, dass das Wählen eines Wertes ähnlich funktioniert, wie die bedingte Ausführung. Auch hier wird Gebrauch von den Eigenschaften der Wahrheitswerte gemacht.

3.3 Natürliche Zahlen

Natürlich benötigt eine Programmiersprache auch Zahlen, einschließlich der Null. Negative Zahlen werden hier nicht behandelt, das heißt, die kleinst mögliche Zahl ist Null.

3.3.1 Darstellung

Da es im Lambda-Kalkül keine Zahlen, wie sie im Allgemeinen verwendet werden, gibt, müssen auch sie durch Terme ausgedrückt werden. Dies geschieht, indem man eine Konstante einführt, die die Null symbolisiert und die in jeder Zahl vorhanden ist. Die Darstellung der Null besteht dann nur aus dieser Konstante. Für alle anderen Zahlen n wird auf der Konstante n mal ein Term ausgeführt [4, S. 60 ff.]:

$$\begin{aligned} c_0 &= \lambda s.\lambda z.z \\ c_1 &= \lambda s.\lambda z.s\, z \\ c_2 &= \lambda s.\lambda z.s(s\, z) \end{aligned}$$

In diesen Termen stellt das z die Konstante für die Null dar. Das s ist der Term, der wiederholt ausgeführt wird. Es sei noch einmal festgehalten, dass hier Repräsentationen der Zahlen verwendet werden. Es handelt sich nicht um Zahlen, sondern um Terme, die sich ebenso verhalten.

3.3.2 Rechenoperationen

Für eine Programmiersprache sind auch Berechnungen notwendig. Es sollen exemplarisch Inkrement, Addition und Multiplikation betrachtet werden. Auch Dekrementierung und Subtraktion sind möglich, werden aber hier nicht behandelt.
Die Formel für Inkrement [3, S. 41] ist noch relativ leicht verständlich:

$$\text{ink} \;= \lambda n.\lambda s.\lambda z.s(n\, s\, z)$$

Zum besseren Verständnis wird aber auch hier wieder ein Beispiel angeführt:

$$\begin{aligned} \text{ink}\; c_1 = \\ (\lambda n.\lambda s.\lambda z.s(n\, s\, z))c_1 \rightarrow \\ (\lambda s.\lambda z.s(c_1\, s\, z)) = \\ (\lambda s.\lambda z.s((\lambda a.\lambda b.a\, b)s\, z)) \rightarrow \\ (\lambda s.\lambda z.s(s\, z)) = \\ c_2 \end{aligned}$$

Man kann erkennen, dass die Formel vor der gegebenen Zahl noch einmal den Term ausführt, der angibt, um wie vieles größer als Null die Zahl ist und so ihren Zweck erfüllt.
Die Addition kann nun mit Hilfe von Inkrement dargestellt werden. Addiert man $n + m$ kann man n auch m mal inkrementieren [4, S. 61]:

$$\text{plus} \;= \lambda m.\lambda n.\lambda s.\lambda z.m\, s(n\, s\, z)$$

Nun kann auch $2 + 1$ bestimmt werden:

$$\begin{aligned}
\text{plus } c_2\, c_1 &= \\
(\lambda m.\lambda n.\lambda s.\lambda z.m\, s(n\, s\, z))c_2\, c_1 &\to \\
\lambda s.\lambda z.c_2\, s(c_1\, s\, z) &= \\
\lambda s.\lambda z.c_2\, s((\lambda a.\lambda b.a\, b)s\, z) &\to \\
\lambda s.\lambda z.(\lambda a.\lambda b.a(a\, b))s(s\, z) &\to \\
\lambda s.\lambda z.s(s(s\, z)) &= \\
c_3 &
\end{aligned}$$

Hier ist besonders die Umbenennung der Variablen in den Zahlendarstellungen zu beachten. Dies geschieht, um Konflikte vorzubeugen.
Multiplikation funktioniert ähnlich wie die Addition. Hier wird für $(m \cdot n)$ m mal n aufaddiert [4, S. 61]:

$$\text{mul } = \lambda m.\lambda n.m(\text{plus } n)c_0$$

Zu Beginn wird einmal c_0 addiert, um das z, also die Variable, die in einer Zahl die Null repräsentiert, zu umgehen. Ansonsten wäre das Ergebnis immer zu groß:

$$\begin{aligned}
\text{mul } c_2\, c_2 &= \\
(\lambda m.\lambda n.m(\text{plus } n)c_0)c_2\, c_2 &\to \\
c_2(\text{plus } c_2)c_0 &= \\
(\lambda s.\lambda z.s(s\, z))(\text{plus } c_2)_0 &\to \\
\text{plus } c_2(\text{plus } c_2\, c_0) &\to \\
\text{plus } c_2\, c_2 &\to \\
c_4 &
\end{aligned}$$

Andere Rechenoperationen und Vergleiche bleiben nun dem Leser überlassen, mögliche Terme finden sich in [4, S. 61 ff.].

3.4 Rekursion

Für eine Programmiersprache ist nun natürlich auch Rekursion wichtig. Ansonsten ist eine unbekannte Anzahl an Wiederholungen schwer möglich. Problematisch ist dabei, dass im Lambda-Kalkül ein Term sich selbst nicht kennt, das heißt, er kann sich auch nicht selbst wieder aufrufen. Dabei ist es allerdings leicht, einen nicht-terminierenden Term zu erstellen:

$$\text{omega } = (\lambda x.xx)(\lambda x.xx)$$

Führt man hier β-Reduktion durch, erhält man wieder den selben Term. Die Rekursion soll allerdings nach einer vorgeschriebenen Anzahl an Wiederholungen zu einem Ende

kommen und es muss einen Term geben, der immer wieder ausgeführt wird [1, S. 109 ff.]. Gibt es nun einen Term *fix*, für den gilt:

$$\text{fix } M \xrightarrow{*} M(\text{fix } M)$$

so kann Rekursion dargestellt werden. Die Terminierung wird dann durch einen Test in M erreicht, der entweder wieder *fix M* ausführt oder einen vorgegebenen Wert zurückliefert. Da das ohne Beispiel schwer zu verstehen ist, folgt hier die Berechnung der Fakultät [4, S. 66 f.]:

$$\begin{aligned} \text{M} &= \lambda f.\lambda n.\text{test (iszero } n)(\lambda x.c_1)(\text{mul } n(f(\text{dec } n))) \\ \text{fac} &= \text{fix } M \end{aligned}$$

Die nicht bekannten Operatoren wie *iszero* können vom interessierten Leser selbst gebildet oder in [1, S. 61 ff.] nachgeschlagen werden. Mit den gerade vorgestellten Formeln für die Fakultät ergibt sich:

$$\begin{aligned} \text{fac } c_1 &= \\ \text{fix } M\ c_1 &\xrightarrow{*} \\ M(\text{fix } M)c_1 &= \\ (\lambda f.\lambda n.\text{test (iszero } n)(c_1)(\text{mul } n(f(\text{dec } n)))\ (\text{fix } M)\ c_1 &\rightarrow \\ \text{test (iszero } c_1)(c_1)(\text{mul } c_1(\text{fix } M\ (\text{dec } c_1))) &\rightarrow \\ \text{mul } c_1(\text{fix } M\ c_0) &\xrightarrow{*} \\ \text{mul } c_1(M(\text{fix } M)\ c_0) &= \\ \text{mul } c_1((\lambda f.\lambda n.\text{test (iszero } n)(c_1)(\text{mul } n(f(\text{dec } n))))(\text{fix } M)\ c_0) &\rightarrow \\ \text{mul } c_1(\text{test (iszero } c_0)(c_1)(\text{mul } n((\text{fix } M(\text{dec } c_0)))) &\rightarrow \\ \text{mul } c_1\ c_1 &\rightarrow \\ c_1 & \end{aligned}$$

Jetzt bleibt nur noch zu zeigen, dass es einen solchen Operator *fix* gibt. Auch hier gibt es wieder verschiedene Möglichkeiten sogenannte Fixpunkt-Kombinatoren zu erstellen. *Fixpunkt* deshalb, weil immer wieder derselbe Term auftritt. Einer der leichtesten ist folgender, der auch Y-Kombinator genannt wird [1, S. 110 ff.] Allerdings ist er nicht immer problemlos verwendbar. Am Rande sei bemerkt, dass bei verschiedenen Auswertungsstrategien des Kalküls Fehler auftreten können. Dies wird hier aber nicht behandelt, kann jedoch in [4, S. 65] nachgelesen werden. Hier nun der Y-Kombinator:

$$\text{fix } = \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$

Führt man *fix* M für einen beliebigen Term M aus, ergibt sich folgende Rechnung:

$$\begin{aligned}
&\text{fix } M = \\
&(\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ M \rightarrow \\
&(\lambda x.M(x\ x))\ (\lambda x.M(x\ x)) \rightarrow \\
&M\ ((\lambda x.M(x\ x))\ (\lambda x.M(x\ x))) \leftarrow \\
&M\ ((\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ M) = \\
&M\ (\text{fix } M)
\end{aligned}$$

Der nach links gerichtete Pfeil macht deutlich, dass es sich um eine umgekehrte β-Reduktion handelt. Hier wird ausgeklammert und nicht eingesetzt. Man kann aber sehen, dass der hier gezeigte Operator alle nötigen Voraussetzungen erfüllt. Damit ist nun auch Rekursion möglich und das Ziel, eine minimale Programmiersprache zu entwerfen, wurde erreicht.

Literatur

[1] Dr. Martin Erwig. *Grundlagen funktionaler Programmierung.* R. Oldenbourg Verlag, München, 1999.

[2] Prof. Dr. Dirk W. Hoffmann. *Theoretische Informatik.* Hanser, München, 2015.

[3] Prof. Dr. Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung.* Springer Verlag, Berlin [u.a.], 2009.

[4] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, Mass.[u.a.], 2002.

[5] Peter Pepper und Petra Hofstedt. *Funktionale Programmierung.* Springer Verlag, Berlin [u.a.], 2006.