

Der untypisierte Lambda-Kalkül

Manuel Eberl
eberlm@cs.tum.edu

19. August 2011

Inhaltsverzeichnis

1	Einleitung	2
2	Grundlagen	2
2.1	Notation	2
2.2	Beispiele zu Abstraktion und Applikation	3
2.3	Umwandlung von Ausdrücken	3
2.3.1	Freie und gebundene Variablen	3
2.3.2	Substitution	4
2.3.3	α -Konversion	5
2.3.4	β -Reduktion	5
2.3.5	β -Normalform	5
2.4	„Nichtterminierende“ Ausdrücke	6
2.5	Currying	6
3	Kodierung von Datentypen	7
3.1	Boole'sche Wahrheitswerte	7
3.2	Paare	8
3.3	Natürliche Zahlen - die Church-Numerale	8
3.4	Arithmetische Operationen	9
3.4.1	Nachfolgeroperation	9
3.4.2	Addition	9
3.4.3	Multiplikation	9
3.4.4	Vorgängerfunktion und Subtraktion	10
3.4.5	Vergleiche	10
4	Rekursive Funktionen	11
5	Zusammenfassung	12

1 Einleitung

Im frühen 20. Jahrhundert versuchten mehrere Mathematiker, grundlegende, abstrakte Systeme zu definieren, auf denen man die komplette Mathematik aufbauen kann. Im Gegensatz zu Systemen wie Zermelo-Fraenkel, das auf Mengen als grundlegende Struktur aufbaut, wurden auch einige Systeme entwickelt, die auf Funktionen aufbauen. Eines dieser Systeme ist der Lambda-Kalkül.

Der Lambda-Kalkül wurde 1928 von Alonzo Church entwickelt und die erste Version 1932 erstmals publiziert. In den darauffolgenden Jahren wurden sowohl in der ursprünglichen Version von 1932 als auch in der daraufhin revidierten Version von Churchs Logik Widersprüche entdeckt (Kleene-Rosser-Paradoxon). 1936 reduzierte Church seine Logik dann schließlich auf das, was er „pure Lambda calculus“ nannte, den Kern seiner Logik, der heute als Lambda-Kalkül bekannt ist.

Ähnlich wie ältere Ansätze von Peano, Schönfinkel und Curry bietet der Lambda-Kalkül die Möglichkeit, Funktionen zu definieren und aufeinander anzuwenden. Die Macht dieses einfachen Systems wurde anfangs deutlich unterschätzt, es hat sich aber gezeigt, dass der Lambda-Kalkül gleichmächtig zur Turing-Maschine ist und damit alle berechenbaren Probleme mit dem Lambda-Kalkül ausgedrückt werden können.

Das Interesse am Lambda-Kalkül und der verwandten kombinatorischen Logik von Curry war über mehrere Jahre hinweg gering. In den 60er-Jahren begannen jedoch Informatiker, sich dafür zu interessieren, was schließlich zur Entwicklung funktionaler Programmiersprachen wie LISP und Haskell führte.

Im Nachfolgenden soll eine kurze Einführung in den untypisierten Lambda-Kalkül gegeben werden.

2 Grundlagen

2.1 Notation

Die Grundlegende Idee des Lambda-Kalküls ist: In einem einzigen Ausdruck werden (anonyme) Funktionen definiert und aufeinander angewendet. Jeder Lambda-Ausdruck ist eine Funktion, die einen Lambda-Ausdruck auf einen Lambda-Ausdruck abbildet.

Es gibt dabei drei grundlegende Arten von Ausdrücken, die verschachtelt werden: *Variablen*, *Funktionsabstraktion* und *Funktionsapplikation*. Sie bilden zusammen die *Lambda-Ausdrücke*.

Definition 1 (Lambda-Ausdruck):

Ein Lambda-Ausdruck sei wie folgt definiert:

Variable Eine Variable x ist ein Lambda-Ausdruck.

Abstraktion Sei x eine Variable und U ein Lambda-Ausdruck. Dann ist die Abstraktion $\lambda x. U$ ein Lambda-Ausdruck.

Applikation Seien U, V Lambda-Ausdrücke. Dann ist die Applikation $U V$ ein Lambda-Ausdruck.

Zusätzlich können noch Klammern zur Gruppierung verwendet werden. Ohne Klammern gelten folgende Regeln:

1. Abstraktionen erstrecken sich so weit nach rechts wie möglich, d.h. $\lambda x. a b c$ entspricht $\lambda x. (a b c)$, nicht etwa $(\lambda x. a) b c$.
2. Die Applikation ist linksassoziativ, d.h. $U V W$ entspricht $(U V) W$, nicht etwa $U (V W)$.
 $U V W$ bedeutet also, ausführlich gesagt, „ U wird auf V angewendet und die entstehende Funktion auf W “, während $U (V W)$ eine einfache Verkettung ist und „ U wird auf das Ergebnis von $V W$ angewendet“ bedeutet.

Als notationelle Vereinfachung können verschachtelte Abstraktionen zusammengefasst werden, d.h. $\lambda x. \lambda y. T$ kann auch folgendermaßen geschrieben werden: $\lambda xy. T$.

Die Bedeutung ist dabei die folgende:

Eine *Abstraktion* $\lambda x. U$ definiert eine anonyme Funktion $x \rightarrow U$. Hierbei ist U ein Ausdruck, der x enthalten kann (dann wird der Parameter der Funktion für x eingesetzt), es aber nicht zwingend muss (dann ist die Funktion konstant). Eine *Applikation* ist die Anwendung einer Funktion auf einen Parameter.

2.2 Beispiele zu Abstraktion und Applikation

Im Folgenden werden einige einfache Funktionen definiert und angewendet, um Abstraktion und Applikation zu zeigen und zu die Anwendung einer Funktion auf eine andere (Funktionen höherer Ordnung) zu verdeutlichen.

Beispiele von einfachen Funktionen

Wir definieren die folgenden Funktionen:

- $id := \lambda x. x$ - erhält ein Argument und gibt dieses zurück (Identität)
- $twice := \lambda f x. f (f x)$ - wendet eine übergebene Funktion zwei mal verkettet auf ihr Argument an.
- sqr : Erhält ein Argument und quadriert dieses.¹

Diese Funktionen lassen sich nun etwa folgendermaßen anwenden:

- $id a$: Liefert a
- $twice id a$: Liefert $id (id a)$, was wiederum a liefert.
- $id id$: Liefert id
- $sqr 3$: Liefert 9
- $twice sqr 3$: Liefert 81
- $twice sqr$: Liefert eine „hoch vier“-Funktion

Hinweis: Der Lambda-Kalkül bietet keine Möglichkeit, Bezeichner für Funktionen zu definieren - die oben verwendeten Namen sind keine symbolischen Bezeichner, sondern Abkürzungen. In Wirklichkeit besteht der Lambda-Ausdruck nur aus einer langen Reihe von Variablen, Abstraktionen und Applikationen - man muss also für jeden dieser Namen immer die komplette, „expandierte“ Form einsetzen, es gibt keine Möglichkeit auf Funktionen per „Namen“ zuzugreifen. Dies wird bei der Rekursion noch wichtig werden.

2.3 Umwandlung von Ausdrücken

2.3.1 Freie und gebundene Variablen

Ähnlich wie bei Ausdrücken mit Quantoren gibt es im Lambda-Kalkül zwei Arten von Variablen: Freie und gebundene. Variablen, die in einer Lambda-Abstraktion definiert werden, heißen im Körper dieser Abstraktion *gebunden*, Variablen, die nicht in einer Abstraktion gebunden sind, heißen *frei*.

¹Definition folgt später, da die Kodierung von Zahlen im Lambda-Kalkül noch nicht eingeführt wurde. Für den Moment nehmen wir einfach an, dass wir Zahlen definieren können und arithmetische Operationen wie sqr auf ihnen durchführen können.

Beispiele:

- $\lambda x.x y$: x ist innerhalb der Abstraktion gebunden, y ist frei
- $\lambda x.\lambda x.x$: das x ist von der inneren Abstraktion gebunden und „verdeckt“ das x der äußeren
- $(\lambda x.x) x$: das erste x ist gebunden, das zweite ist frei.
- $(\lambda x.\underline{x}) (\lambda y.\underline{x})$: das erste unterstrichene x ist gebunden, das zweite unterstrichene x ist frei.

Eine genauere Definition des Begriffs der freien Variablen ist wie folgt:

Definition 2 (Freie Variablen):

$FV(A)$ bezeichne die Menge der freien Variablen eines Ausdrucks A . Diese Menge wird wie folgt rekursiv definiert:

- Für eine Variable x ist $FV(x) = \{x\}$.
- Für einen Lambda-Ausdruck B ist $FV(\lambda x.B) = FV(B) \setminus \{x\}$
- Für Lambda-Ausdrücke B, C ist $FV(B C) = FV(B) \cup FV(C)$

Ein Lambda-Ausdruck ohne freie Variablen heißt *geschlossener Lambda-Ausdruck* oder *Kombinator*.

2.3.2 Substitution

Wichtig für die weiteren Umformungen ist die Substitution, d.h. das Ersetzen einer Variable durch einen Ausdruck.

Definition 3 (Substitution):

Die Substitution $[x := T]$ von x durch T sei folgendermaßen definiert:

- $x[x := R] = R$
- $a[x := R] = a$ (für $x \neq a$)
- $(S T)[x := R] = S[x := R] T[x := R]$
- $(\lambda x.T)[x := R] = \lambda x.T$ (keine Substitution, da x hier gebunden ist)
- $(\lambda a.T)[x := R] = \lambda a.T[x := R]$ (für $x \neq a$ und a nicht frei in R , „freshness condition“)

Die letzte Bedingung (a nicht frei in R) ist sehr wichtig, man sagt auch „ a muss frisch in R sein“, da es sonst bei Ersetzungen zu Problemen kommt - denn sobald R eingesetzt wird, sind die a in R nicht mehr frei, sondern werden als das gebundene a interpretiert („capturing“).

Beispiel: Sei $A := \lambda a.a \text{ narf } foo$. Wir wollen nun $A[foo := bar \ a \ wuppd]$ berechnen - also alle Vorkommnisse von foo durch den Ausdruck $bar \ a \ wuppd$ ersetzen. a ist aber nicht frisch in $bar \ a \ wuppd$ - es kommt dort bereits frei vor.

Eine einfache Ersetzung würde zu dem Ausdruck $\lambda a.a \text{ narf } (bar \ a \ wuppd)$ führen, in dem das a in den Klammern das gebundene a ist - nicht, wie gewünscht das freie. Deshalb ist die Bedingung der Frische wichtig.

In der momentanen Definition ist die Substitution also nur partiell definiert. Falls die freshness condition nicht erfüllt ist, so muss a in der Abstraktion zuerst durch α -Konversion in eine frische Variable umbenannt werden, um das capturing zu vermeiden. Damit ist die Substitution dann total definiert.

2.3.3 α -Konversion

Ähnlich wie bei den Quantoren sind die Namen gebundener Variablen irrelevant. Die Lambda-Ausdrücke $\lambda x. x$ und $\lambda y. y$ sind völlig gleichwertig - sie liefern die gleichen Ergebnisse. Den Vorgang, eine gebundene Variable in einer Abstraktion umzubenennen, nennt man α -Konversion.

Definition 4 (α -Konversion):

Die α -Konversion einer Abstraktion $\lambda x. S$ für Variablen x, y und einem Lambda-Ausdruck S mit y frisch in S sei wie folgt definiert:

$$\lambda x. S \stackrel{\alpha}{\equiv} \lambda y. S[x := y]$$

Ausdrücke, die sich durch α -Konversion ineinander umwandeln können, nennt man α -äquivalent.

Die Forderung, dass y frisch in S verhindert, dass wir zur Auflösung des Problems wieder eine α -Konversion benötigen.

Mit dieser Definition können wir nun auch das Problem von oben lösen:

$$\begin{aligned} & (\lambda a. a \text{ narf } foo)[foo := bar \ a \ wuppd\ i] \stackrel{\alpha}{\equiv} \\ & \stackrel{\alpha}{\equiv} (\lambda a'. a' \text{ narf } foo)[foo := bar \ a \ wuppd\ i] = \lambda a'. a' \text{ narf } (bar \ a \ wuppd\ i) \end{aligned}$$

2.3.4 β -Reduktion

Die β -Reduktion ist eine Auflösung einer Applikation. Hierfür definieren wir zuerst den Begriff des Redex (reducible expression):

Definition 5 (Redex):

Ein Ausdruck der Form $(\lambda x. S) T$ mit einer Variablen x und Lambda-Ausdrücken S, T heißt *Redex* (reducible expression). Ein Ausdruck, der keinen Redex enthält, heißt *irreduzibel*.

Die β -Reduktion ist dann eine Operation, die bei einem Redex wie eben definiert den Parameter T in S für x einsetzt. Dies kann mithilfe der Substitution folgendermaßen ausgedrückt werden:

Definition 6 (β -Reduktion):

Die β -Reduktion eines Redex $(\lambda x. S) T$ sei wie folgt definiert:

$$(\lambda x. S) T \xrightarrow{\beta} S[x := T]$$

2.3.5 β -Normalform

Bei vielen Lambda-Ausdrücken erhält man nach endlich vielen Durchführungen der β -Reduktion einen irreduziblen Ausdruck. Einen solchen Ausdruck nennt man β -Normalform. Eine interessante Eigenschaft der β -Reduktion ist ihre *Konfluenz*, d.h. wenn ein Ausdruck A eine β -Normalform besitzt, so sind alle möglichen irreduziblen Ausdrücke, die man durch mehrmalige β -Reduktion von A erreichen kann, zu dieser α -äquivalent; Es gibt also maximal eine β -Normalform, egal, in welcher Reihenfolge man ableitet.

Man kann durch ungeschickte Auswertungsreihenfolge auch bei einem Ausdruck, der eine Normalform besitzt, unendlich lange reduzieren, ohne zur Normalform zu kommen (siehe nächster Abschnitt). Falls zwei verschiedene Abläufe von Reduktionen eines Ausdrucks aber zu einer Normalform führen, so ist diese immer (bis auf α -Konversion) gleich.

2.4 „Nichtterminierende“ Ausdrücke

Da der Lambda-Kalkül gleichmächtig zur Turingmaschine ist, muss es in ihm aber auch Ausdrücke geben, die nicht „terminieren“. Das heißt in diesem Fall, dass sie keine β -Normalform besitzen - man kann durch endlich viele Reduktionen keinen irreduziblen Ausdruck erhalten. Der Ausdruck kann bei fortschreitender Reduktion unter Umständen sogar immer länger werden.

Ein klassisches Beispiel ist der sogenannte Ω -Kombinator:

$$\omega := \lambda x. x x$$

$$\Omega := \omega \omega = (\lambda x. x x) (\lambda x. x x)$$

Die β -Reduktion von Ω führt wieder zu Ω . In Anlehnung daran kann man sich auch einfach einen Ausdruck überlegen, der unendlich weit wächst:

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

Reduktion führt zu:

$$(\lambda x. x x x) (\lambda x. x x x) \Rightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \Rightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \Rightarrow \dots$$

Ein Beispiel für einen wie im vorherigen Abschnitt angesprochenen Ausdruck, der zwar eine Normalform besitzt, aber bei ungünstiger Auswertungsreihenfolge nicht zur Normalform führt, wäre folgender:

$$false \ \Omega \ id = (\underline{\lambda ab. b}) ((\underline{\lambda x. x x}) (\lambda x. x x)) (\lambda x. x)$$

Wird der Redex mit der ersten unterstrichenen Abstraktion reduziert, so reduziert sich der gesamte Ausdruck in einem Schritt zu $id = \lambda x. x$. Wird jedoch der Redex mit der zweiten unterstrichenen Abstraktion reduziert, so ist das Ergebnis wieder der Ausdruck selbst und man kann dies unendlich oft wiederholen, ohne die Normalform zu erreichen. Die Auswertungsreihenfolge hat also keinen Einfluss auf das Ergebnis, aber sie kann sehr wohl beeinflussen, *ob überhaupt* ein Ergebnis gefunden wird.

Vor allem bei rekursiven Funktionen muss man daher eine „lazy evaluation“-Strategie wählen, d.h. nur wirklich benötigte Ergebnisse werden auch berechnet, ansonsten würden rekursive Ausdrücke nie terminieren. Dann wird die Normalform immer erreicht - falls sie existiert. Die Frage, ob eine Normalform existiert, ist wegen der Äquivalenz zum Halteproblem nicht berechenbar.

2.5 Currying

Wie vorher bereits erwähnt ist ein Lambda-Ausdruck eine Funktion, die einen Lambda-Ausdruck auf einen Lambda-Ausdruck abbildet. Funktionen mit mehreren Parametern gibt es im Lambda-Kalkül direkt nicht - es ist aber möglich, dies indirekt zu realisieren: Will man etwa eine Funktion *add* definieren, die zwei Parameter *a* und *b* entgegennimmt und addiert, so fasst man *add* so auf, dass *add a* eine Funktion add_a zurückgibt, die einen Parameter *b* entgegennimmt und $a + b$ zurückgibt. *add a b* reduziert sich dann zu $add_a b$ und dies wiederum zum Ergebnis von $a + b$.

Der Ausdruck *add 4* liefert also eine „Addiere 4“-Funktion, die einen Parameter erwartet und zu diesem 4 addiert. *add 4 38* ist dann nichts anderes als eine Anwendung dieser „Addiere 4“-Funktion auf 38, und der entstehende Ausdruck reduziert sich zu 42.

Dies kann allgemein mit jeder Funktion *f* mit *n* Parametern gemacht werden: Man wandelt *f* in eine Funktion um, die ihren ersten Parameter a_1 auf eine Funktion f'_{a_1} abbildet. Diese wiederum bildet den zweiten Parameter a_2 auf f'_{a_1, a_2} ab und so weiter, bis man eine Funktion $f'_{a_1, a_2, \dots, a_{n-1}}$ erhält, die den letzten Parameter a_n auf das Ergebnis $f'_{a_1, a_2, \dots, a_{n-1}, a_n} = f(a_1, a_2, \dots, a_{n-1}, a_n)$ abbildet.

Dies nennt man *Currying*

Definition 7 (Funktionen mit mehreren Parameter - Currying):

Es sei f eine Funktion mit n Parametern:

$$f : A_1 \times A_2 \times \dots \times A_n \rightarrow B$$

Diese Funktion kann per Currying folgendermaßen als Funktion höherer Ordnung dargestellt werden:

$$f' : A_1 \rightarrow (A_2 \rightarrow (\dots (A_n \rightarrow B)))$$

Beispiel:

Wir betrachten eine Funktion *reverse*, die zwei Funktionen f und g als Parameter nimmt und $g f$ zurückgibt - also f auf g anwendet.

$$\text{reverse} := \lambda fg. g f$$

Wenden wir nun *reverse* auf zwei Parameter a und b an, so erhalten wir:

$$\text{reverse } a b = (\lambda fg. g f) a b \Rightarrow (\lambda g. g a) b = \text{reverse}'_a b \Rightarrow b a$$

Im ersten Reduktionsschritt erhalten wir also $\text{reverse}'_a$, eine Funktion, die einen Parameter g erwartet und diesen auf a anwendet. Im zweiten Reduktionsschritt erhalten wir dann das Ergebnis $b a$.

3 Kodierung von Datentypen

Die einzigen Strukturen, die im Lambda-Kalkül existieren, sind Variablen und Funktionen. Es stellt sich die Frage, wie man damit geläufige Datentypen wie Booleans und Zahlen oder auch komplexere Datentypen wie Listen darstellen kann. Prinzipiell wird dies dadurch erreicht, dass man diese Datentypen als Funktionen mit einem gewissen Verhalten definiert und dann Operationen auf diesen Datentypen über Anwendung dieser Funktionen realisiert. Das einfachste Beispiel hierfür sind wohl Booleans.

3.1 Boole'sche Wahrheitswerte

Es gibt zwei Boole'sche Wahrheitswerte - *true* und *false* - und vier „interessante“ Operationen auf ihnen - das unäre *not* und die binären *and*, *or* und *xor*. Die einfachste Kodierung für *true* und *false* ist die als eine Funktion, die zwei Parameter enthält und einen von ihnen zurückgibt. *true* gibt hierbei immer den ersten zurück, *false* immer den zweiten. Die Definition von *true* und *false* ist daher sehr einfach und intuitiv:

Definition 8 (true und false):

$$\text{true} := \lambda ab. a$$

$$\text{false} := \lambda ab. b$$

Hiermit lassen sich bereits direkt „if-then-else“-Verzweigungen realisieren: Betrachten wir den Ausdruck $a U V$, wobei a ein Boole'scher Wahrheitswert wie oben definiert ist und U und V beliebige Lambda-Ausdrücke. Falls $a = \text{true}$ ist, so wird V verworfen und der Ausdruck reduziert sich zu U , falls $a = \text{false}$ ist wird U verworfen und der Ausdruck reduziert sich zu V . Der Ausdruck $a U V$ entspricht also direkt der Verzweigung **if a then U else V** ;

Unter Verwendung dieser „if“-Verzweigung können auch sehr intuitiv alle Boole'schen Operatoren definiert werden. Ein *not* ist ein einfaches **if a then false else true** ;

Ein *and* kann man sich ähnlich leicht überlegen: Will man a *and* b berechnen, so kann man sich zuerst a ansehen. Ist $a = \text{false}$, so steht das Ergebnis bereits fest - a *and* b muss *false* sein. Ist $a = \text{true}$, so hängt das Ergebnis allein von b ab - der Gesamtausdruck ist genau dann *true*, wenn auch b *true* ist. Es ergibt sich also **if a then b else false** ;

Wir können also nun unsere vier interessanten Boole'schen Operationen im Lambda-Kalkül definieren:

Operator	if-then-else	Lambda-Ausdruck
<i>not</i>	if a then false else true;	$\lambda a. a \text{ false true}$
<i>and</i>	if a then b else false;	$\lambda ab. a \text{ b false}$
<i>or</i>	if a then true else b;	$\lambda ab. a \text{ true b}$
<i>xor</i>	if a then not b else b;	$\lambda ab. a \text{ (not b) b}$

3.2 Paare

Die Kodierung von Paaren (2-Tupeln) ist sehr intuitiv: Man überlegt sich, welche Eigenschaften ein Paar haben sollte - es sollte zwei Lambda-Ausdrücke enthalten und man sollte auf selbige zugreifen können. Eine Möglichkeit, dies nun als Funktion zu kodieren, ist folgende:

Ein Paar wird als Funktion aufgefasst, die eine andere Funktion z erwartet und diese auf ihre beiden Parameter anwendet, d.h. ein Paar $p = (a; b)$ bildet eine Funktion z ab auf $z a b$.

Als nächstes stellt sich die Frage, wie man am einfachsten auf diese Werte zugreifen kann. Wir benötigen Funktionen *first* und *second*, die ein Paar $p = (a; b)$ jeweils auf a bzw. b abbilden. Dazu müssen wir das Paar p auf eine Funktion anwenden, die genau das tut, also im Fall von *first* müssen wir p auf eine Funktion anwenden, die zwei Parameter erwartet und den ersten zurückgibt - eine solche Funktion haben wir bereits definiert, es ist die *true*-Funktion. Das gleiche gilt analog für *second* und *false*. Der Ausdruck $p \text{ true}$ liefert also a und $p \text{ false}$ liefert b .

Definition 9 (Paare):

Ein Paar $p = (a; b)$ wird kodiert als $\lambda z. z a b$.

Der Zugriff auf Paare erfolgt mit folgenden Funktionen:

$\text{first} := \lambda p. p \text{ true}$

$\text{second} := \lambda p. p \text{ false}$

3.3 Natürliche Zahlen - die Church-Numerale

Die Kodierung natürlicher Zahlen ist etwas schwieriger. Church verwendete hierzu die nach ihm benannten Church-Numerale: Eine natürliche Zahl n wird kodiert als eine Funktion, die eine andere Funktion n mal verkettet und dann auf ihr Argument anwendet², d.h. $n : f x \rightarrow f^n x$, oder, in Lambda-Schreibweise:

Definition 10 (Church-Numerale):

Das Church-Numeral, das $n \in \mathbb{N}_0$ kodiert, ist:

$$\lambda fx. \underbrace{f(f \dots (f x) \dots)}_{n \text{ mal}} = \lambda fx. f^n x$$

Beispiele:

$0 \hat{=} \lambda fx. x$ (bildet jede Funktion auf *id* ab und entspricht zufällig genau *false*)

$1 \hat{=} \lambda fx. f x$

$2 \hat{=} \lambda fx. f (f x)$

...

²Das Church-Numeral 2 entspricht daher genau der zuvor im Beispiel verwendeten *twice*-Funktion.

3.4 Arithmetische Operationen

3.4.1 Nachfolgeroperation

Die wohl grundlegendste Operation für das Arbeiten mit natürlichen Zahlen ist die Nachfolgeroperation $\text{succ} : n \rightarrow n + 1$.

Sie ist relativ einfach im Lambda-Kalkül umzusetzen: n ist eine Funktion, die f und x auf $f^n x$ abbildet, dann muss der Nachfolger von n eine Funktion sein, die f und x auf $f^{n+1} x$ abbildet. Wir berechnen also zuerst $f^n x$, was einfach $n f x$ ist, und wenden dann f noch einmal auf das Ergebnis an.

Definition 11 (Nachfolgeroperation - succ):

$$\text{succ} := \lambda n f x. f (n f x)$$

Beispiel:³

$$\begin{aligned} \text{succ } 1 &= \text{succ } (\lambda f x. f x) = \underline{(\lambda n f x. f (n f x))} \underline{(\lambda f x. f x)} \Rightarrow \\ &\Rightarrow \lambda f x. f \underline{((\lambda f' x'. f' x') f x)} \Rightarrow \\ &\Rightarrow \lambda f x. f (f x) = 2 \end{aligned}$$

3.4.2 Addition

Die Addition $m + n$ lässt sich ähnlich einfach auf zwei Arten realisieren: Eine Möglichkeit ist, die Nachfolgerfunktion einfach m mal auf n anzuwenden, also $m \text{ succ } n$, die entstehenden Ausdrücke werden beim praktischen Berechnen von Hand allerdings etwas unhandlich, weshalb hier eine andere Methode verwendet werden soll.

Wir berechnen einfach $n f x$ - also $f^n x$ - und setzen das Ergebnis in f^m ein. Es ergibt sich f^{m+n} .

Definition 12 (Addition):

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

Beispiel:

$$\begin{aligned} + 2 3 &= \underline{(\lambda m n f x. m f (n f x))} \underline{2} \underline{3} \Rightarrow \\ &\Rightarrow \lambda f x. 2 f (3 f x) = \\ &= \lambda f x. \underline{(\lambda f' x'. f' (f' x'))} \underline{f} \underline{((\lambda f' x'. f' (f' (f' x')))) f x} \Rightarrow \\ &\Rightarrow \lambda f x. (\lambda x'. f (f x')) \underline{((\lambda f' x'. f' (f' (f' x')))) f x} \Rightarrow \\ &\Rightarrow \lambda f x. \underline{(\lambda x'. f (f x'))} \underline{(f (f (f x)))} \Rightarrow \\ &\Rightarrow \lambda f x. f (f (f (f (f x)))) = 5 \end{aligned}$$

3.4.3 Multiplikation

Zur Berechnung der Multiplikation $m \cdot n$ überlegen wir uns, was für eine Funktion $m f$ ist. $m f$ ist eine Funktion, die einen Parameter x erwartet und $f^m x$ zurückgibt. In der Funktion stehen also, einfach gesagt, $m f$'s hintereinander. Was passiert, wenn nun n auf diese Funktion angewandt wird? Die Funktion wird wiederum n mal verkettet, das heißt wir haben nun n mal jeweils $m f$'s hintereinander in der Funktion stehen und die Funktion bildet daher x auf $f^{m \cdot n} x$ ab, was genau die Multiplikation ist. Wir müssen also nur $n (m f)$, oder analog $m (n f)$ berechnen.

³Der erste unterstrichene Teil ist immer die in diesem Schritt reduzierte Abstraktion, die folgenden unterstrichenen Teile die eingesetzten Parameter

Definition 13 (Multiplikation):

$$* := \lambda mnf.m (n f)$$

3.4.4 Vorgängerfunktion und Subtraktion

Das Finden eines Vorgängers eines Church-Numerals ist im Lambda-Kalkül deutlich schwieriger als das Finden des Nachfolgers, denn eine Funktion ein Mal öfter anzuwenden ist einfach - ein Mal weniger jedoch nicht. Der Einfachheit halber definieren wir im Folgenden $pred\ 0 = 0$, oder anders gesagt: $pred\ n := n \dot{-} 1$

$\dot{-}$ steht für die saturierte Subtraktion, d.h. $m \dot{-} n = \max(m - n, 0)$. Ist also $m < n$, so ist $m \dot{-} n = 0$, da wir negative Zahlen vermeiden wollen, ansonsten ist $m \dot{-} n = m - n$.

Die Peano-Axiome fordern eigentlich, dass die 0 keinen Vorgänger hat, aber es ist in diesem Zusammenhang nützlich, dies so zu definieren.

Um nun den Vorgänger einer Zahl n zu berechnen, definieren wir eine Hilfsfunktion Φ , die folgende Abbildung eines Paares p auf ein anderes Paar realisiert:

$$\Phi : (a; b) \longrightarrow (a + 1; a)$$

$$\Phi := \lambda pz.z (succ (first\ p)) (first\ p)$$

Wird diese Funktion auf $(0; 0)$ angewendet, so erhält man $(1; 0)$, wendet man sie auf $(1; 0)$ an, so erhält man $(2; 1)$ und so weiter. Diesem Schema folgend erhält man bei n verketteten Anwendungen der Funktion auf das Paar $(0; 0)$ als Ergebnis das Paar $(n; n \dot{-} 1)$. Der gesuchte Vorgänger ist also das zweite Element dieses Paares. Es ergibt sich:

$$pred := \lambda n.second (n\ \Phi\ \lambda z.z\ 0\ 0)$$

Analog zur ersten vorgestellten Variante der Addition lässt sich nun die (saturierte) Subtraktion $m \dot{-} n$ ausdrücken als n -malige Anwendung der Vorgängerfunktion auf m :

$$\dot{-} := \lambda mn.n\ pred\ m$$

3.4.5 Vergleiche

Die grundlegendste Vergleichsfunktion ist der Test auf Gleichheit mit 0. Hieraus lassen sich später alle anderen Vergleichsoperatoren aufbauen. Um ein Church-Numeral mit 0 zu vergleichen, nutzen wir aus, dass für jede Funktion f gilt: $0\ f = id$

Wir können nun also eine Hilfsfunktion h konstruieren, die einen Parameter x erwartet und konstant *false* zurückgibt. Wenden wir dann n auf h an, so kann man zwei Fälle unterscheiden:

- Ist $n = 0$, so ist $n\ h = id$ und $n\ h$ gibt einfach seinen Parameter zurück.
- Ist $n \neq 0$, so verwirft $n\ h$ seinen Parameter und gibt *false* zurück.

Wenn wir also h n -mal verketteten und ihm *true* als Parameter liefern, so erhalten wir genau die gewünschte Funktion *iszero*.

Definition 14 (Test auf 0 - iszero):

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

Für unsere eigentlichen Vergleichsoperatoren betrachten wir nun zunächst den \leq -Operator und formen um:

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0^4$$

Wir können also \leq auf *iszero* und $\dot{-}$ reduzieren und durch Vertauschen der Parameter erhalten wir auch gleich noch den \geq -Operator, durch Negation der beiden $>$ und $<$ und durch Kombination von \geq und \leq auch $=$.

Definition 15 (Vergleichsoperatoren):

$$\leq := \lambda mn. \text{iszero } (\dot{-} m n)$$

$$\geq := \lambda mn. \text{iszero } (\dot{-} n m)$$

$$> := \lambda mn. \text{not } (\leq m n)$$

$$< := \lambda mn. \text{not } (\geq m n)$$

$$= := \lambda mn. \text{and } ((\leq m n) (\geq m n))$$

4 Rekursive Funktionen

Eine Rekursion mit fester Rekursionstiefe wurde bereits implizit eingeführt - jede Anwendung eines Church-Numerals n auf eine Funktion f wendet f n -mal rekursiv auf sich selbst an (siehe z.B. *pred*). Dies gleicht allerdings eher einer iterativen bzw. primitiv-rekursiven Vorgehensweise (vgl. LOOP-Berechenbarkeit). Interessant wären aber rekursive Funktionen mit variabler Abbruchbedingung. Als Beispiel soll folgende rekursive Definition der Fakultät $n!$ dienen:⁵

$$n! := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Oberflächlich betrachtet führt dies zu folgendem Lambda-Ausdruck:

$$\text{fac_rec} := \lambda n. (\text{iszero } n) 1 (* n (\text{fac_rec } (\text{pred } n)))$$

Hierbei ergibt sich aber ein Problem: *fac_rec* wird in seiner eigenen Definition verwendet. Zu diesem Zeitpunkt ist es aber ja noch gar nicht definiert. Würde man versuchen, sämtliche Funktionen, die im obigen Ausdruck nur mit ihrem Namen abgekürzt sind, tatsächlich aufzuschreiben, so müsste man den kompletten Ausdruck wieder für *fac_rec* einsetzen, worin wieder der Ausdruck *fac_rec* vorkommt, dieser müsste wieder expandiert werden und so weiter - der Ausdruck wäre also in seiner expandierten Form unendlich groß.

Das Problem ist also prinzipiell, dass *fac_rec* sich nicht selbst „kennt“ und sich daher nicht selbst aufrufen kann. Man kann nun einfach einen weiteren Parameter zu *fac_rec* hinzufügen - nämlich *fac_rec* selbst. Das neue *fac_rec* erhält einfach eine Funktion f als Parameter, von der sie annimmt, dass es sie selbst ist. Anstatt nun sich selbst direkt aufzurufen, ruft die Funktion diesen Parameter f auf und übergibt ihm zusätzlich zu dem normalen Parameter wiederum f .

Wir definieren nun also *fac_rec* folgendermaßen:

$$\text{fac_rec} := \lambda fn. (\text{iszero } n) 1 (* n (f f (\text{pred } n)))$$

Um eine „schönere“ Funktion zu erhalten, können wir nun noch eine Hilfsfunktion *fac* definieren, die diese Eigenheit der *fac_rec*-Funktion verbirgt:

$$\text{fac} := \lambda n. \text{fac_rec } \text{fac_rec } n$$

⁴siehe Definition der saturierten Subtraktion

⁵Die Fakultät ist natürlich auch primitiv-rekursiv bzw. LOOP-berechenbar, man könnte sie also auch auf die gleiche Art und Weise wie die Vorgängerkfunktion mit einer rekursiven Hilfsfunktion wie dem dort verwendeten Φ definieren

Zum Abschluss soll nun noch einmal *fac 3* als kompletter Lambda-Ausdruck ohne abkürzende Namen gezeigt werden:

$$\begin{aligned}
 & (\lambda n. (\lambda f n_1. ((\lambda n_2. n_2 (\lambda x. (\lambda ab. b)) (\lambda ab. a)) n_1) (\lambda f x. f x) ((\lambda m n_2 f. m (n_2 f)) n_1 (f f ((\lambda n_3. \\
 & (n_3 (\lambda p z. z ((\lambda n_4 f x. f (n_4 f x))(p (\lambda ab. a))) (p (\lambda ab. a))) (\lambda z. z (\lambda ab. b) (\lambda ab. b))) (\lambda ab. b)) n_1)))) \\
 & (\lambda f n_1. ((\lambda n_2. n_2 (\lambda x. (\lambda ab. b)) (\lambda ab. a)) n_1) (\lambda f x. f x) ((\lambda m n_2 f. m (n_2 f)) n_1 (f f ((\lambda n_3. (n_3 (\lambda p z. z \\
 & ((\lambda n_4 f x. f (n_4 f x))(p (\lambda ab. a))) (p (\lambda ab. a))) (\lambda z. z (\lambda ab. b) (\lambda ab. b))) (\lambda ab. b)) n_1)))) n) \lambda f x. f (f (f x))
 \end{aligned}$$

Dieser Ausdruck reduziert sich schließlich zu:

$$\lambda f x. f (f (f (f (f x)))) = 6$$

5 Zusammenfassung

Der untypisierte Lambda-Kalkül ist, ähnlich wie die Turing-Maschine, ein minimales, abstraktes Modell für Berechenbarkeit und ist daher für Beweise in der theoretischen Informatik sehr nützlich. Er ist jedoch völlig ungeeignet dafür, tatsächlich zum Programmieren benutzt zu werden. Das größte Problem ist jedoch das Fehlen jeglicher Typen - eine Funktion für Boole'sche Werte kann auch auf Church-Numerale oder sogar auf arithmetische Operationen angewandt werden. Geschieht soetwas versehentlich, so entstehen schwer nachvollziehbare Fehler. Allgemein ist die Durchführung von Berechnungen im untypisierten Lambda-Kalkül kompliziert und umständlich - ähnlich wie die Turing-Maschine ist es nur als abstraktes Modell gedacht, es konkret für Berechnungen anzuwenden, ist nicht sehr sinnvoll.

Diese Probleme lassen sich allerdings leicht beheben, indem Typen für Booleans, Zahlen, Listen usw. eingeführt werden, anstatt die umständliche Kodierung zu verwenden. Werden dann noch einige andere Veränderungen durchgeführt, um den Kalkül leichter handhabbar zu machen, und einiger „syntaktischer Zucker“ hinzugefügt, so erhält man eine vollständige funktionale Programmiersprache wie LISP oder Haskell. Aufgrund ihrer nützlichen Eigenschaften haben Lambda-Ausdrücke oder davon abgeleitete funktionale Konzepte sogar Einzug in imperative/objektorientierte Programmiersprachen wie Python, Scala und indirekt sogar C++ gehalten.