Programmieren mit dem untypisierten Lambda-Kalkül

Die Ausarbeitung zum Proseminar

Chaoran Chen

18. Oktober 2014

Inhaltsverzeichnis

1 Einleitung **2**
- 1.1 Lambda-Kalkül als Modell der rein funktionalen Programmierung . . . 2
- 1.2 Rein funktionale Programmierung . . . 2

2 Der Lambda-Kalkül **3**
- 2.1 Abstraktion . . . 3
- 2.2 Applikation . . . 4
- 2.3 Schreibweisen . . . 4
- 2.4 Alpha-Konversion . . . 4
- 2.5 Beta-Reduktion . . . 4

3 Programmieren mit dem Lambda-Kalkül **5**
- 3.1 Booleans . . . 5
- 3.2 Paare / Tupel . . . 6
- 3.3 Natürliche Zahlen . . . 6
- 3.4 Rekursion . . . 7

Literatur **9**

1 Einleitung

1.1 Lambda-Kalkül als Modell der rein funktionalen Programmierung

Der Lambda-Kalkül, entworfen in den 1930er Jahren von Alonzo Church, ist ein Modell der rein funktionalen Programmierung. Es ist eine formale, mathematische Sprache, die die Arbeitsweise der funktionalen Programmierung modelliert und damit das, was die Turingmaschine für imperative Programmiersprachen ist. Als Kalkül dient er der formalen Ableitung von Aussagen und Formulierung von Beweisen und ermöglicht eine einfache Sicht auf das Paradigma.

1.2 Rein funktionale Programmierung

Das Paradigma der funktionalen Programmierung sieht vor, dass ein Programm *nur* aus deterministischen Funktionen besteht – das heißt, dass die Ausgabe nur von der Eingabe abhängt. Es gibt keine äußeren Zustände, die Einfluss auf die Funktionen nehmen. Im Gegensatz zur imperialen Programmierung gibt es damit keine veränderbare Variablen und damit auch keine Zustände, sondern nur Werte, Konstanten und Parameter.

Werte sind in der funktionalen Programmierung Funktionen. Diese können eine, mehrere oder keine Parameter erfordern. Sie geben immer genau einen Wert zurück.

Konstanten sind vordefinierte Werte, Beispiele wären die natürlichen Zahlen oder die arithmetischen Funktionen. Die Menge der vorgegebenen Konstanten unterscheiden sich zwischen den Sprachen. Die in der Praxis eingesetzten Sprachen besitzen oft einen großen Umfang an Konstanten, unterscheiden sich jedoch in ihrer Mächtigkeit nicht von einer Sprache ohne vordefinierte Konstanten.

Parameter sind innerhalb von Funktionen zu finden. Über diese werden einer Funktion beim Aufruf Argumente – das heißt, Konstanten oder Werte – übergeben.

Viele funktionale Programmiersprachen besitzen entgegen des Paradigmas nicht nur deterministische Funktionen, sondern auch die Möglichkeit mit Zuständen zu arbeiten. Dies ist auch erforderlich, wenn zum Beispiel Benutzereingaben eingelesen oder Zufallszahlen generiert werden sollen. Eine funktionale Programmiersprache gilt dann als *rein*, wenn sie keine Elemente erlaubt, die dem Paradigma widersprechen.

In dem folgenden Abschnitt 2 werden die Bestandteile und Funktionsweise des Kalküls beschrieben. Im Abschnitt 3 wird anschließend die Mächtigkeit des Lambda-Kalküls anhand von Modellierungen wichtiger Elemente einer Programmiersprache aufgezeigt.

2 Der Lambda-Kalkül

Wie oben beschrieben, gibt es in der rein funktionalen Programmierung nur Werte, Konstanten und Parameter. Damit ist ein Lambda-Term t wie gefolgt definiert:

$t ::= x$	Wert / Variable
$\mid a$	Konstante
$\mid \lambda x.\ t$	Abstraktion / Lambda-Funktion,
$\mid$	mit x als Parameter und t als Rumpf
$\mid t\ t$	Applikation

Ein Lambda-Kalkül heißt *rein*, wenn er keine Konstanten enthält.

2.1 Abstraktion

Abstraktionen sind anonyme Funktionen, sie sind also namenlos. Für eine übersichtliche Darstellung werden sie im Folgenden zwar Bezeichnungen erhalten, diese dienen aber nur als Platzhalter. Ein Aufruf anderer Funktionen wie in anderen Programmiersprachen ist nicht möglich. Was das in der Praxis bedeutet, wird bei Rekursionen im Abschnitt 3.4 beschrieben.

Es wird zwischen *freie* und *gebundene* Variablen unterschieden. Es gilt: eine Variable x ist frei, falls sie in keiner Abstraktion steht, wo x als Parameter angegeben wird. So sind in der folgenden Abstraktion

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

das zweite y (durch das erste y) und das zweite x (durch das erste x) gebunden und das dritte y frei.

Funktionen im Lambda-Kalkül haben genau einen Parameter. Um dennoch Funktionen zu definieren, die „typischerweise“ zwei oder mehrere Parameter besitzen, werden Abstraktionen verwendet, die genau eine weitere Abstraktion im Rumpf besitzen.

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

So kann eine Funktion für das Prüfen zweier Werte x und y auf ihre Gleichheit konstruiert werden, indem zuerst mit der Übergabe von x eine Funktion „istGleichX“ definiert und anschließend der Funktion y übergeben wird.

2.2 Applikation

Entsprechend anderer Programmiersprachen wird auch im Lambda-Kalkül zwischen der Definition und Applikation von Funktionen unterschieden. Bei letzterer wird der erstgenannte Term (der Operator) auf den zweitgenannten Term (der Operand bzw. das Argument) angewendet.

2.3 Schreibweisen

Für eine einfache, lesbare Darstellung, gelten die folgende Schreibweisen:

$$\begin{aligned} \lambda x.\ (\lambda y.\ t) &\equiv \lambda x\ y.\ t \\ \lambda x.\ (\lambda y.\ (\lambda z.\ t)) &\equiv \lambda x.\ \lambda y.\ \lambda z.\ t \quad \text{(rechtsassoziativ)} \\ (t_1\ t_2)\ t_3 &\equiv t_1\ t_2\ t_3 \qquad\quad \text{(linksassoziativ)} \end{aligned}$$

2.4 Alpha-Konversion

Die Alpha-Konversion einer Abstraktion ist eine Umbenennung der Parameter und der Variablen, dabei werden alle zugehörige Vorkommnisse ersetzt.

Beispiele:

$$\begin{aligned} \lambda x.\ (x\ (\lambda x.\ x)) &\to_\alpha \lambda y.\ (y\ (\lambda x.\ x)) \\ \lambda x.\ (x\ y) &\to_\alpha \lambda x.\ (x\ z) \end{aligned}$$

Durch eine Alpha-Konversion dürfen keine freie Variablen gebunden werden:

$$\lambda x.\ (x\ y) \not\to_\alpha \lambda x.\ (x\ x)$$

2.5 Beta-Reduktion

Die Beta-Reduktion ist eine Substitution und wird bei der Applikation angewendet. Vereinfacht dargestellt gilt:

$$(\lambda x.\ t)\ a \to_\beta t[x \mapsto a]$$

In Worten: Bei der Applikation $((\lambda x.\ t)\ a)$ wird a für den Parameter x eingesetzt, dabei werden alle Vorkommnisse von x in t durch a ersetzt. Auch hier dürfen keine freie Variablen gebunden werden, ggf. müssen zuerst Alpha-Konversionen durchgeführt werden.

Beispiel:

$$(\lambda x\ y.\ x)\ y \twoheadrightarrow_\beta \lambda y.\ y$$

Stattdessen:

$$\begin{aligned}(\lambda x\ y.\ x)\ y &\to_\alpha (\lambda x\ y'.\ x)\ y \\ &\to_\beta \lambda y'.\ y\end{aligned}$$

Ein Lambda-Term ist in der *Normalform*, wenn er nicht mehr reduziert werden kann. Nicht alle Terme haben eine Normalform:

$$\begin{aligned}(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) &\to_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ &\to_\beta \ldots\end{aligned}$$

Es gilt die *Church/Rosser-Eigenschaft*: Zwei Lambda-Terme t_1, t_2, die über die Alpha-Konversion und Beta-Reduktion ineinander transformiert werden können, gelten als *äquivalent*. Für diese gibt es ein t mit $t_1 \to^* t$ und $t_2 \to^* t$. Daraus folgt auch, dass ein Lambda-Term nicht mehr als eine Normalform haben kann.

3 Programmieren mit dem Lambda-Kalkül

Als Modell der funktionalen Programmierung ist der Lambda-Kalkül so mächtig wie die in der Praxis verwendeten Sprachen und gleichzeitig auch Turing-vollständig. In diesem Abschnitt werden die Booleans, die Paare, die natürlichen Zahlen und die Rekursion am Beispiel der Fakultätsfunktion modelliert.

Die gezeigten Funktionen sind mögliche Kodierungen, die im Zusammenspiel mit anderen Funktionen die jeweiligen geforderten Eigenschaften besitzen. Es sind weder die einzigmöglichen Kodierungen noch haben sie an sich eine semantische Bedeutung – so kann eine „Plus"-Funktion der natürlichen Zahlen auch für gänzlich andere Zwecke verwendet werden.

3.1 Booleans

Booleans werden verwendet, um Verzweigungen zu steuern. Eine Lambda-Abstraktion würde also als Kodierung für $true$ bzw. $false$ fungieren können, wenn man gleichzeitig eine Abstraktion if findet, die bei $true$ den ersten Zweig und bei $false$ den zweiten Zweig zurückgibt. Dies ist bei den folgenden Termen der Fall:

$$\begin{aligned}true &:= \lambda t\ f.\ t \\ false &:= \lambda t\ f.\ f \\ if &:= \lambda l\ m\ n.\ l\ m\ n\end{aligned}$$

Ein Beispiel, um zu sehen, dass es funktioniert:

$$\begin{aligned} if\ true\ x\ y &\equiv (\lambda l\ m\ n.\ l\ m\ n)\ (\lambda t\ f.\ t)\ x\ y \\ &\rightarrow_\beta (\lambda m\ n.\ (\lambda t\ f.\ t)\ m\ n)\ x\ y \\ &\rightarrow_\beta (\lambda n.\ (\lambda t\ f.\ t)\ x\ n)\ y \\ &\rightarrow_\beta (\lambda t\ f.\ t)\ x\ y \\ &\rightarrow_\beta (\lambda f.\ x)\ y \\ &\rightarrow_\beta x \end{aligned}$$

Weitere Funktionen im Zusammenhang mit Booleans:

$$\begin{aligned} and &:= \lambda x\ y.\ x\ y\ false \\ or &:= \lambda x\ y.\ x\ true\ y \\ not &:= \lambda x.\ x\ false\ true \\ eq &:= \lambda x\ y.\ x\ (y\ true\ false)\ (y\ false\ true) \end{aligned}$$

3.2 Paare / Tupel

Paare / 2-Tupel:

$$\begin{aligned} pair &:= \lambda f\ s\ b.\ b\ f\ s \\ first &:= \lambda p.\ p\ true \\ second &:= \lambda p.\ p\ false \end{aligned}$$

n-Tupel:

$$\begin{aligned} nTupel &:= \lambda x_1\ \ldots\ x_n\ b.\ b\ x_1\ \ldots\ x_n \\ iItem &:= \lambda p.\ p\ (\lambda x_1\ \ldots\ x_n.\ x_i) \end{aligned}$$

Eine Liste kann als verschachtelte Paare kodiert werden.

3.3 Natürliche Zahlen

Eine bekannte Kodierung für die natürlichen Zahlen sind die Church Numerale c_0, c_1, c_2, ...:

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

Eine Zahl n ist damit als eine Funktion definiert, die zwei Parameter s und z besitzt und s n-Mal auf z anwendet. Auf der Idee basierend kann eine Funktion $isZero$ konstruiert werden, indem einer Zahl für s und z Terme übergeben werden, sodass genau dann, falls s im Rumpf auftaucht, $false$ zurückgegeben wird.

$$isZero := \lambda m.\ m\ (\lambda x.\ false)\ true$$

Bei der Nachfolger-Funktion wird einer Zahl c_n zuerst das λ entfernt und eine neue Zahl c_{n+1} zurückgegeben, bei der das erste Argument s einmal öfter auf z angewendet wird als bei c_n. Die Funktionen für die Addition, Multiplikation und Potenz können mit demselben Prinzip konstruiert werden.

$$\begin{aligned} successor &:= \lambda n\ s\ z.\ s\ (n\ s\ z) \\ plus &:= \lambda m\ n\ s\ z.\ m\ s\ (n\ s\ z) \\ times &:= \lambda m\ n\ s.\ m\ (n\ s) \\ power &:= \lambda m\ n.\ m\ n \end{aligned}$$

Bei der Vorgänger-Funktion muss ein s im Rumpf entfernt werden. Dies wird dadurch erreicht, dass nacheinander Paare von Zahlen beginnend mit (c_0,c_0), (c_0,c_1) bis (c_{n-1},c_n) berechnet werden, um am Ende bei dem letzten Paar das erste Element zurückzugeben. zz und ss sind hierbei Hilfsfunktionen.

$$\begin{aligned} zz &:= pair\ c_0\ c_0 \\ ss &:= \lambda p.\ pair\ (second\ p)\ (successor\ (second\ p)) \\ predecessor &:= \lambda m.\ first\ (m\ ss\ zz) \\ minus &:= \lambda m\ n.\ n\ predecessor\ m \end{aligned}$$

Mit der Definition der Subtraktion kann nun leicht die Gleichheitsfunktion konstruiert werden:

$$eq := and\ (isZero\ (minus\ m\ n))\ (isZero\ (minus\ n\ m))$$

3.4 Rekursion

Da es in der funktionalen Programmierung keine Zustände gibt, sind Iterationsschleifen, die in imperativen Programmiersprachen verwendet werden, nicht möglich. Stattdessen werden Rekursionen verwendet, bei der eine Funktion sich selber wieder aufruft. Dies ist im Lambda-Kalkül jedoch problematisch insofern, dass die Funktionen anonym sind und sich nicht zur Laufzeit aufrufen können. Wie das Problem dennoch gelöst werden kann, wird abschließend im Folgenden am Beispiel der Fakultätsfunktion gezeigt.

Zuerst soll eine Funktion fac' definiert werden, die für ihre Funktionalität die Übergabe der Fakultätsfunktion benötigt.

$$fac' := \lambda f\ n.\ (isZero\ n)\ c_1\ (times\ n\ (f\ (predecessor\ n)))$$

Anschließend wird ein *Fixpunktkombinator* fix gesucht, mit der Eigenschaft

$$fix\ t \rightarrow_\beta t\ (fix\ t).$$

Einen möglichen Fixpunktkombinator wäre:

$$fix := (\lambda x\ f.\ f\ (x\ x\ f))(\lambda x\ f.\ f\ (x\ x\ f))$$

Beispiel:

$$\begin{aligned} fix\ t \equiv\ & (\lambda x\ f.\ f\ (x\ x\ f))(\lambda x\ f.\ f\ (x\ x\ f))\ t \\ \rightarrow_\beta\ & (\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ t \\ \rightarrow_\beta\ & t\ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t) \\ \equiv\ & t\ (fix\ t) \end{aligned}$$

Somit kann die Fakultätsfunktion wie gefolgt definiert werden:

$$faculty := fix\ fac'$$

Auch hierfür ein Beispiel:

$$\begin{aligned} faculty\ c_1 \equiv\ & fix\ fac'\ c_1 \\ \rightarrow_\beta\ & fac'\ (fix\ fac')\ c_1 \\ \equiv\ & (\lambda f\ n.\ (isZero\ n)\ c_1\ (times\ n\ (f\ (prd\ n))))(fix\ fac')c_1 \\ \rightarrow_\beta^2\ & (isZero\ c_1)\ c_1\ (times\ c_1\ ((fix\ fac')\ (prd\ c_1))) \\ \rightarrow_\beta^*\ & times\ c_1\ ((fix\ fac')\ c_0) \\ \rightarrow_\beta\ & times\ c_1\ (fac'\ (fix\ fac')\ c_0) \\ \rightarrow_\beta^*\ & times\ c_1\ c_1 \\ \rightarrow_\beta^*\ & c_1 \end{aligned}$$

Literatur

[1] Marco Block und Adrian Neumann. *Haskell-Intensivkurs: Ein kompakter Einstieg in die funktionale Programmierung*. Xpert.press. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2011. ISBN: 978-3-642-04717-6.

[2] H. A. Klaeren und Michael Sperber. *Die Macht der Abstraktion: Einführung in die Programmierung*. 1. Aufl. Leitfäden der Informatik. Wiesbaden: B.G. Teubner Verlag / GWV Fachverlage, Wiesbaden, 2007. ISBN: 978-3-8351-0155-5.

[3] Kenneth C. Louden. *Programmiersprachen: Grundlagen, Konzepte, Entwurf*. 1. Aufl. Informatik-Lehrbuch-Reihe. Bonn und Albany [u.a.]: Internat. Thomson Publ., 1994. ISBN: 9783929821031.

[4] Tobias Nipkow. *Lambda-Kalkül*. 2004. URL: `http://www4.informatik.tu-muenchen.de/lehre/vorlesungen/logik/WS0708/lambda.pdf`.

[5] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. ISBN: 9780262256810.