Untypisiertes Lambda Kalkül und ein Programm zum Ermitteln einer optimalen Normalform

verfasst von
Nils Jakubzick

betreuende Lehrkraft: Herr Steinfeld

Abgabedatum: 05.07.2024

1. Einleitung

Programmiersprachen waren für mich schon immer faszinierend. Seit ich anfing zu programmieren, habe ich immer liebend gerne neue Sprachen und Paradigmen ausprobiert, um dabei herauszufinden, was mir daran gefällt oder auch nicht gefällt. Dabei sind mir vor Allem funktionale Programmiersprachen ans Herz gewachsen, die zwar nicht immer für das Erstellen von realen Anwendungen geeignet sind, aber trotzdem mit ihrem puren Code (d.h. ohne Seiteneffekte) und in ihrer deklarativen Art eine Art Eleganz aufweisen, die ich faszinierend fand und finde.

Im Rahmenplan für Informatik in Mecklenburg-Vorpommern aus dem Jahr 2019 ist die Turing-Maschine fest verankert [3]. Ich halte das für eine gute Idee, denn die Turing-Maschine eignet sich aufgrund ihrer Anschaulichkeit gut für den Unterricht. Sie besteht aus mehreren Komponenten, die auch in der Realität so existieren könnten (das Eingabeband kann natürlich nicht unendlich groß sein) und arbeitet nach einer festen, sowie relativ simplen Schrittfolge. Trotzdem ist sie heute noch für moderne Computer und ihre Mächtigkeit, also was sie denn berechnen können, relevant. Der Grund hierfür liegt vor allem an der Einfachheit, mit der Beweise, sowie Komplexitätsbetrachtungen durchgeführt werden können. Selbst relativ komplexe Sachverhalte, wie das Halteproblem können auf diese Weise den Schülern nahegebracht werden. Es gibt also gute Gründe, die Turing-Maschine im Unterricht zu behandeln.

Es gibt aber auch andere Modelle der theoretischen Informatik, die dieselbe Mächtigkeit aufweisen, wie Turing-Maschinen - sie sind also auch Turing-Vollständig. Und obwohl diese anderen Modelle, zu denen vor allem der Lambda-Kalkül und die partiell rekursiven Funktionen gehören, weniger anschaulich sind, macht sie das nicht weniger interessant. Besonders der Lambda-Kalkül ist spannend, weil es die Grundlage für die erste und damit auch für alle nachfolgenden funktionalen Programmiersprachen bildet.

In dieser Facharbeit sollen die Funktionsweise sowie einige Konsequenzen aus der relativ simplen Definition des Lambda-Kalküls erläutert werden. Um einen Ausdruck “optimal” zu reduzieren, muss man “optimal” zuerst definieren. Weiterhin wird erklärt, wie man einen Ausdruck optimal reduziert. Zuletzt soll das selbstgeschriebene Programm *Blis* vorgestellt werden, welches einen von mir vorgeschlagenen Dialekt des Lambda-Kalküls erkennen und optimal reduzieren kann. Der Dialekt dient dabei lediglich der Vereinfachung des Syntax.

2. Lambda Kalkül

2.1. Geschichte

Der Lambda-Kalkül entstand 1932-1933 aus Alonzo Church's Versuch, die Fundamente der Mathematik theoretisch zu begründen. Es war hierbei Teil einer größeren Theorie, in der auch beispielsweise mehr Symbole definiert wurden. In 1935 extrahierte er eine erste Version von dem, was später als der Lambda-Kalkül bekannt wurde [4]. Die endgültige Version wurde in seiner Arbeit aus dem Jahr 1936 veröffentlicht. Die letztgenannte Veröffentlichung enthält auch erste Ansätze der späteren Church-Turing-These [2]. Kleene zeigte 1936, dass der Lambda-Kalkül und die partiell rekursiven Funktionen äquivalent sind [4]. Im selben Jahr veröffentlichte Turing seine Arbeit zu Turing-Maschinen und zeigt dabei auch, dass letztere mit dem Lambda-Kalkül äquivalent sind [4].
In der zweiten Hälfte der 1950er Jahre entwickelte John McCarthy LISP [5]. Man findet einige gemeinsame Eigenschaften der Sprachen. Das Reduzieren eines Ausdrucks zu einer simpleren Form, Funktionen als Variablen und die Definition von Funktionen über die λ-Notation sind in beiden Sprachen vorhanden. Weiterhin nennt McCarthy explizit Church's Lambda-Kalkül als Inspiration für seine Funktionsdefinition [5]. LISP hat als erste funktionale Sprache einen enormen Einfluss auf die weitere Entwicklung von Programmiersprachen und wird zusammen mit C als eine der beiden reinen Programmiersprachen beschrieben [6].

2.2. Definition

In seinem Paper "An Unsolvable Problem of Elementary Number Theory" definiert Church den untypisierten Lambda-Kalkül. Dies stellt eine Verallgemeinerung seines vorher definierten typisierten Lambda-Kalküls dar [4]. Der typisierte Lambda-Kalkül ist dabei nicht nur komplizierter programmatisch zu reduzieren, sondern ist auch nicht Turing-Vollständig und damit weniger mächtig als sein untypisiertes Gegenstück [1]. Aus diesen Gründen befasst sich diese Facharbeit ausschließlich mit der untypisierten Variante.
Untypisiertes Lambda Kalkül (weiterhin: Lambda Kalkül) besteht aus den Symbolen '{', '}', '[', ']', '(', ')' sowie 'λ' [2]. Hierbei erfüllen die verschiedenen Arten der öffnenden und schließenden Klammern aber jeweils denselben Zweck. Das bedeutet, eine Definition ausschließlich mit runden Klammern würde nichts an ihrer Allgemeingültigkeit verlieren. Aus diesem Grund werden in dieser Arbeit ausschließlich runde Klammern verwendet.
Variablen werden definiert als die kleingeschriebenen Buchstaben des lateinischen Alphabets [2]. Daher beinhaltet der Ausdruck '*ab*' nicht eine Variable *ab*, sondern die zwei Variablen *a* und *b*.

Weiterhin werden neben den Variablen zwei weitere Sprachkonstrukte definiert, die heutzutage als *Abstraktion* bzw. *Applikation* bekannt sind [2]. Church definiert in seiner Arbeit zwar auch die Umkehrung der Applikation. Diese soll aber hier keine weitere Relevanz finden. Die Abstraktion hat die Form $\lambda x.\, M$, wobei M ein beliebiger λ-Ausdruck sein kann. Die Applikation hat die Form $(N\ M)$ wobei *N* und *M* beliebige λ-Ausdrücke sein können. Im Bereich der modernen Informatik ist es anschaulicher, wenn man die Abstraktion als Funktionsdefinition und die Applikation als Funktionsaufruf ansieht. In der Abstraktion $\lambda x.\, M$ würde *x* den Parameter beschreiben und *M* den Funktionskörper. In der Applikation $(N\ M)$ wäre *N* die Funktion und *M* das Argument. Zuletzt ist die Applikation linksassoziativ:

$$((N\ M)\ O)\ \Leftrightarrow\ (N\ M\ O)$$

So können bereits sämtliche Ausdrücke des Lambda Kalküls gebildet werden. Allerdings müssen wir noch definieren, wie die Umwandlung dieser Lambda-Ausdrücke ineinander funktioniert. Hierfür sind insbesondere die α-Umwandlung, sowie die β-Reduktion relevant [2]. Die α-Umwandlung ist verwandt mit dem Konzept, dass Funktionen unabhängig von ihren Parametern definiert sein sollten, also auch mit unterschiedlichen Parameternamen dieselbe Funktion beschreiben sollten. Daher beschreiben $f(x) = x$, $f(y) = y$ und $f(\alpha) = \alpha$ dieselbe, lineare Funktion mit Anstieg 1. Äquivalent dazu sollten auch Funktionen im Lambda Kalkül unabhängig von ihren Parameternamen sein. Die dazugehörige Umwandlung nennt man α-Umwandlung. Es gilt

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

Hierbei bedeutet $M[x \to y]$, dass alle Instanzen von *x* in *M* mit *y* ersetzt wurden. Man sagt "$\lambda x.\, M$ und $\lambda y.\, M[x \to y]$ sind α-äquivalent", weil sie durch α-Umwandlung ineinander umgewandelt werden können. Bei dieser Art der Umwandlung gibt es nur eine Einschränkung. Folgendes Beispiel soll der Verdeutlichung dienen:

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

Hierbei wäre eine α-Umwandlung $[x \to y]$ problematisch, weil *y* im Term $(x\ y)$ bereits eine andere Bedeutung hat. Daher darf die Variable, durch die ersetzt wird, in *M* nicht auftauchen. Ansonsten kann es zu solchen Komplikationen der Variablen kommen. Man kann also sagen, dass α-Umwandlung eine Substitution von Variablen ist, wobei die Variable durch die ersetzt wird, frei sein muss.
Die β-Reduktion ist die andere relevante Form der Umwandlung. Für sie gilt:

$$(\lambda x.M)N \;=\; M[x \to N]$$

Es werden also alle Instanzen von *x* in *M* durch *N* ersetzt. Hierbei ist $(\lambda x.M)N$ □*-äquivalent* zu $M[x \to N]$. Auch bei dieser Art der Umwandlung kann es zu Komplikationen der Variablen kommen. Diese können aber durch α-Reduktion aufgelöst werden, indem die kollidierenden Variablen umbenannt werden. Ist für einen Term keine □-Reduktion mehr möglich, dann ist dieser Term *vollständig* □*-reduziert* beziehungsweise er hat seine *vollständig* □*-reduzierte Normalform* erreicht. Dieser Zustand ist derjenige, der durch das Programm erzeugt werden soll. Es gibt Terme, für die keine vollständige □-Reduktion möglich ist.
In Churchs Abhandlung werden die Begriffe *freie Variable*, *gebundene Variable* sowie *wohlgeformt* definiert [2]. Für diese Begriffe soll eine intuitive Definition genügen. *Wohlgeformt* bedeutet, dass der Ausdruck keine syntaktischen Fehler beinhaltet. Ausdrücke, die nicht gut gebildet sind, sind nicht Bestandteil dieser Facharbeit. *Gebundene Variablen* in einem Term *M* sind solche, die in den umgebenden Funktionsdefinitionen als Argument auftauchen. Im Term $\lambda x.M$ ist *x* innerhalb von *M* gebunden. Variablen, die nicht gebunden sind, sind *frei*.

2.3. Herleitung einfacher Sprachkonstrukte

Die Einfachheit des Lambda Kalküls erschwert ein intuitives Verstehen der Mächtigkeit dieses Werkzeugs. Aus diesem Grund sollen hier einige Sprachkonstrukte, die man aus typischen Programmiersprachen kennt, hergeleitet werden. Eine der größten Sprünge ist der, von den Ein-Argument-Funktionen zu Multi-Argument-Funktionen. Wenn Funktionen im Lambda-Kalkül nur ein einziges Argument haben, wie sollen dann Funktionen, die zwei oder mehr Argumente annehmen, wie beispielsweise das Addieren zweier Zahlen realisiert werden? Die Antwort auf diese Frage ist ein Konzept, welches noch heute in vielen funktionalen Programmiersprachen verwendet wird. Currying, benannt nach Haskell Curry, beschreibt ein Prinzip, bei dem Funktionen andere Funktionen zurückgeben. Spezifischer können so die Argumente nacheinander abgearbeitet werden. Beim Addieren würde beispielsweise die Funktion zum Addieren eine Zahl annehmen und eine Funktion zurückgeben, die wieder eine Zahl annimmt und dann diese beiden Zahlen zusammen addiert. Im Lambda-Kalkül sähe das dann beispielsweise so aus (auch wenn der Lambda-Kalkül natürlich kein Konzept von herkömmlichen Zahlen oder dem Infix-Additions-Operator hat):

$$(\lambda x.\lambda y.(x\ +\ y))\ 3\ 5$$

Also sind Multi-Argument-Funktionen trotz der Beschränkungen des Lambda-Kalküls möglich. Auch Datentypen kann man innerhalb dieses simplen Gerüsts emulieren. Booleans

beispielsweise können durch eine Funktion dargestellt werden, die drei Argumente annimmt: den Wert, wenn der Boolean “Wahr” ist und den Wert, wenn der Boolean “Falsch” ist.

$$true = \lambda w.\lambda f(w)$$
$$false = \lambda w.\lambda f(f)$$

Mit diesen Definitionen kann man bereits Logikgatter, wie zum Beispiel so simulieren:

$$and = \lambda x.\lambda y.(x\ y\ FALSE)$$
$$or = \lambda x.\lambda y.(x\ t\ y)$$
$$not = \lambda x.(x\ FALSE\ TRUE)$$

Aber auch komplexere Datentypen als nur Booleans sind möglich. Beispielsweise ein sogenanntes Pair - ein Datentyp, der zwei Werte enthält.

$$pair = \lambda x.\lambda y.\lambda f(f\ x\ y)$$

Dieses Pair würde man dann mit zwei Werten “beladen” und die Funktion *f* würde von diesen beiden Werten einen zurückgeben. Dies erlaubt aber bereits das Erstellen von Linked Lists. Ist nämlich der zweite Wert eines Pairs, wieder ein Pair, so kann man beliebige Mengen an Daten speichern.
Natürliche Zahlen werden nach der sogenannten Church-Notation definiert. Andere Zahlentypen können natürlich auch definiert werden, dies soll aber nicht Bestandteil dieser Arbeit sein. Hier die ersten 4 natürlichen Zahlen in Church-Notation:

$$0 = \lambda f.\lambda x.x$$
$$1 = \lambda f.\lambda x.f(x)$$
$$2 = \lambda f.\lambda x.f(f(x))$$
$$3 = \lambda f.\lambda x.f(f(f(x)))$$

Das Muster sollte offensichtlich sein. Die Zahl korrespondiert mit der Anzahl von Funktionsaufrufen von *f* auf *x.* Vergleicht man die Definitionen der Zahl *0*, sowie des Booleans *false,* so fällt auf, dass sie α-Äquivalent sind. Dies ist nicht nur analog zur Repräsentation der beiden Werte in modernen Programmiersprachen, sondern zeigt auch eine wichtige Eigenschaft des *untypisierten* Lambda-Kalküls. Da es keine Typen gibt, können dieselben Ausdrücke in verschiedenen Kontexten unterschiedliche Dinge bedeuten. Daher muss man aufpassen, dass man Ausdrücke unterschiedlicher “Datentypen” nicht miteinander vermischt.

Der erste Schritt zur Arithmetik ist die Addition und der erste Schritt zur Addition ist eine Nachfolger-Funktion. Intuitiv würde sie eine Zahl in Church-Notation, sowie eine Funktion annehmen und dann einfach den Funktionsaufruf zurückgeben. Dies führt aber nicht zum Ziel:

Definition der Nachfolger-Funktion *s* (engl. *successor)*

$$s = \lambda n.\lambda f.f(n)$$

Beispielaufruf mit der Zahl zwei und einer beliebigen Funktion

$$s\ 2\ g$$

$$(\lambda n.\lambda f.f(n))\ 2\ g$$

□-Reduktion

$$g(2)$$

Einsetzen der Definition für *2*

$$g(\lambda f.\lambda x.f(f(x)))$$

$$\neq$$

$$g(g(g(x)))$$

Stattdessen führt die folgenden Definition zum Ziel:

$$\lambda n.\lambda f.\lambda x.f(n\ f\ x)$$

Der innere Aufruf *n f x* entfernt sozusagen das *f*- sowie *x*-Argument und lässt nur die (in diesem Fall 2) Aufrufe von *f* auf *x* zurück. Zu diesen Aufrufen wird ein weiterer hinzugefügt und der ganze Ausdruck wird anschließend wieder mit den Argumenten *f* und *x* versehen.
Die weiteren Herleitungsschritte sähen dann in etwa so aus, dass man eine Funktion definiert, die auf Gleichheit mit 0 testet. Zusammen mit einer Subtraktion kann man so Vergleichsoperatoren definieren. Definiert man dann noch Rekursion, beispielsweise über den Y-Kombinator, sowie eine Wenn-Dann-Funktion, so erhält man eine vollständige, funktionale Programmiersprache. Die Programmiersprache ist zwar unpraktisch und sehr ineffizient, aber sie erfüllt alle Anforderungen.

2.4. Über Optimalität

Der Lambda-Kalkül ist Turing-Vollständig [4] und kann daher dieselbe Menge an Funktionen berechnen wie Turing-Maschinen. Daher muss es innerhalb des Lambda Kalkül eine Äquivalenz zum Halteproblem geben. Also gibt es auch in Lambda Kalkül Funktionen, die man nicht

berechnen kann, bzw. solche, die nie vollständig reduziert werden können. Ein Beispiel für eine solche Funktion wäre der sogenannte Ω-Kombinator, der die folgende Struktur hat:

$$\Omega = (\lambda x.\,(x\ x))(\lambda x.\,(x\ x))$$

Ein Versuch der Reduktion sähe dann so aus:

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

Alpha-Umwandlung

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

Einsetzen von $(\lambda y.\,(y\ y))$ für x

$$(\lambda y.\,(y\ y))(\lambda y.\,(y\ y))$$

Alpha Umwandlung

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

Daher reduziert der Kombinator zu sich selbst und eine vollständige Reduktion ist niemals möglich. Das “Halteproblem” des Lambda-Kalküls ist also, ob ein Term vollständig reduziert werden kann oder nicht. Dass diese Frage im allgemeinen Fall nicht trivial ist, merkt man an folgendem Beispiel:

$$((\lambda x.\,(\lambda y.\,y))\ \Omega)\ 5$$

Der Term beinhaltet zwar das unreduzierbare *Ω*, aber das Argument, für welches *Ω* übergeben wird, wird nie benutzt. Daher reduziert sich der Term trotzdem ganz normal zu *5*. Um also herauszufinden, ob eventuell unreduzierbare Terme überhaupt benötigt werden, muss - intuitiv gesehen - der Term evaluiert werden. Church selbst hat nicht gezeigt, dass Lambda Kalkül ein Äquivalent zum Halteproblem beinhaltet - Turing hatte seinen Beweis über das Halteproblem zu diesem Zeitpunkt noch nicht veröffentlicht - sondern benutzte Gödels Beitrag zur Lösung des Entscheidungsproblems. Er wies jedem Term innerhalb des Lambda Kalkül eine Gödel-Zahl zu und beweist auf diese Weise, dass das Entscheidungsproblem innerhalb des Lambda Kalkül unlösbar ist [2].

2.5 Einfluss des Lambda-Kalküls auf Programmiersprachen

Eine wichtige Eigenschaft von funktionalen Programmiersprachen ist, dass sie “pur” sein wollen. Das bedeutet, dass Funktionen bestimmte Argumente besitzen und *ausschließlich* aufgrund dieser Parameter ihr Ergebnis berechnen. Funktionen haben auch keine Seiteneffekte, daher gibt es keinen veränderbaren Zustand außerhalb der Funktion (und manchmal gar keine veränderbaren Zustände überhaupt). Dabei ist der Lambda-Kalkül die Basis aller dieser Sprachen [7] und hat dementsprechend dieselben Eigenschaften. Weiterhin gibt es in der funktionalen Welt auch das Konzept der sogenannten Closures. Das sind Funktionen, welche auf Zustand außerhalb ihrer Funktion zugreifen können. Auch dieses Konzept kommt aus dem Lambda-Kalkül und wird dort aber implizit vorausgesetzt. In Church’s Modell sind weiterhin sämtliche Funktionen anonym, daher besitzen sie keinen Namen. Funktionale Programmiersprachen haben oft ein Konzept für solche Funktionen ohne Namen und orientieren sich in ihrer Notation sogar an Church’s λ-Notation. Beispiele für solche Sprachen wären beispielsweise Lisp, Scheme, Haskell und sogar das imperative Python. Auch das Konzept von Funktionen als Variablen ist selbst für moderne Programmiersprachen nicht selbstverständlich. Dies ist im Lambda-Kalkül offensichtlich möglich, Funktionen sind die einzigen Werte, die Variablen annehmen können, und dementsprechend findet man dieses Konzept in allen funktionalen Programmiersprachen.
Auch wenn der Lambda-Kalkül für funktionale Programmiersprachen essentiell ist, so nehmen doch immer mehr imperative Programmiersprachen funktionale Features an. In neueren Versionen von Sprachen, wie Java, C# oder auch C++ findet man heutzutage viele dieser Eigenschaften wieder. Dies zeigt, wie groß der Einfluss des Lambda-Kalküls auch auf moderne Programmiersprachen ist. Dabei sind diese “Eigenschaften” im Lambda-Kalkül nicht explizit definiert und sind stattdessen Folge einer Definition von Funktionen, die einfacher nicht sein könnte.

3. Erläuterungen zum Programm

3.1. Notation

Andere Programme zur Reduzierung von Ausdrücken des Lambda-Kalküls orientieren sich oft stark an der mathematischen Notation. Lediglich, dass das klein-griechische λ auf einer Tastatur nicht ganz einfach zu schreiben ist, wird manchmal bedacht. In diesen Fällen wird dann, angelehnt an Haskell’s Funktionsnotation, auch ein einfaches ‘\’ anstelle des λ’s erlaubt. Die durch mich vorgeschlagene Notation orientiert sich mehr an den Sprachen der C-Familie, da ich die Funktionsblöcke durch runde Klammern eingrenze. Ein Funktionsaufruf wird durch eine

Funktionsdefinition, gefolgt von einem Punkt und einem Parameter gekennzeichnet. Diese Notation ist nicht nur einfacher zu parsen, sondern bietet auch mehr Ergonomie beim Erstellen eines Ausdrucks. Weil man bei den meisten Ausdrücken des Lambda-Kalküls sowieso Klammern benutzen muss, um den Ausdruck zu disambiguieren, ergibt es Sinn, den Syntax der Abstraktion direkt über die Klammern zu definieren. Weiterhin ist das Benutzen eines Punktes statt Leerzeichen für die Applikation vertrauter für die meisten Programmierer, weil derselbe Syntax für Methodenaufrufe benutzt wird. Auch sorgt diese Entscheidung dafür, dass Leerzeichen beim Parsen von Ausdrücken ignoriert werden können, was das Parsen vereinfacht.
Weiterhin sind aber die vielen Einrückungen, die durch die Klammern entstehen, nicht immer einfach zu lesen. Deshalb gibt es folgende syntaktische Vereinfachung in meinem Dialekt.

$f(g(M)) \Leftrightarrow f,g(M)$ für einen beliebigen Lambda-Ausdruck *M*

Sprachkonstrukt	originales Lambda-Kalkül	vorgeschlagener Dialekt
Abstraktion	$\lambda a.\,\lambda b.\,a$	`a,b(a)`
Applikation	$f\ a\ b$	`f.a.b`

Boolean	$(\lambda a.\,\lambda b.\,\lambda f.\,(f\ a\ b))\ t\ f$	`a,b,f(`
		`  f.a.b`
		`).t.f`

Nachfolger-Funktion	$\lambda n.\,\lambda f.\,\lambda x.\,f(n\ f\ x)$	`n,f,x(`
		`  f.(n.f.x)`
		`)`

Auch wenn Church in seiner ursprünglichen Definition nur Ein-Buchstaben-Variablen vorsah, sind diese, wenn man tatsächlich etwas Nützliches mit der Lambda Kalkül durchführen möchte, eher unpraktisch. Daher erlaubt mein Dialekt auch längere Variablennamen, aus beliebigen Zeichen, die keine Sonderzeichen im Sinne meines Dialekts sind. Daher sind Punkte, Kommata und runde Klammern ausgeschlossen. Zuletzt sieht Lambda Kalkül keinerlei Variablenzuweisungen vor. Da diese aber durchaus emuliert werden können, definieren wir eine *let*-Anweisung, wie folgt:

Ausdruck	Kompiliert zu
`let n M;N`	`n(N).M`

für beliebige Lambda-Ausdrücke *M*, *N* und einen beliebigen Variablennamen n

Dies bindet *n* zu *M* innerhalb von *N* und fungiert so als Variablendefinition. Mehrere Variablendefinitionen pro Programm sind möglich.

3.2. Reduktionsreihenfolge

Die Existenz des Ω-Kombinators eröffnet eine nicht-triviale Frage: In welcher Reihenfolge sollten die Terme reduziert werden? Fängt man beispielsweise einmal an, Ω zu reduzieren, so gerät man automatisch in eine Endlosschleife. Trotzdem kann es ja sein, dass Ω höchstens als unbenutztes Argument auftaucht. In diesem Fall wäre ein zu frühes Reduzieren von Ω fatal, weil man so keine vollständig □-reduzierte Normalform erreichen könnte. Aber etwas Anderes wäre noch viel fataler: Ist es möglich, dass bestimmte Umformungsschritte dazu führen, dass die vollständig □-reduzierte Normalform nicht mehr erreichbar ist? Können wir uns also, wenn wir nicht aufpassen, in eine “Sackgasse” manövrieren? Auf beide Fragen liefert das Church-Rosser-Theorem die passende Antwort.

Es ist nicht möglich, dass nach einer gewissen Abfolge von Schritten die vollständig □-reduzierte Normalform nicht mehr erreichbar ist. Genauer gesagt besagt ein Teil des Church-Rosser-Theorems, dass wenn von Anfangszustand *A* sowohl die Zustände E_1 und E_2 erreichbar sind, dann existiert ein Zustand *E*, der von beiden Zuständen E_1 und E_2 aus erreichbar ist [2].

Es existieren mehrere Reduktionsreihenfolgen von denen zwei hier näher betrachtet werden sollen. Gegeben ist eine Funktion mit einem dazugehörigen Argument, die reduziert werden sollen. Nach der Definition der □-Reduktion müssten wir jetzt alle Instanzen der Variable der Funktion durch das Argument ersetzen. Dieses Vorgehen nennt man *Normal Order Reduction* [8]. Auf der anderen Seite wird das Argument innerhalb der Funktion wahrscheinlich mehrmals auftauchen. Wenn wir uns dazu entscheiden, das Argument einzusetzen und es existiert mehrmals, dann muss es mehrmals reduziert werden. Es könnte also auch Sinn ergeben, erst das Argument zu reduzieren, bevor es eingesetzt wird. Dieses Vorgehen nennt man *Applicative Order Reduction* [8]. Obwohl Applicative Order in der Realität Vorteile innerhalb der Performance bringt, garantiert es *nicht* das Erreichen der Normalform. Man beachte folgendes Beispiel: das Argument könnte ein nicht-reduzierbarer Term, wie beispielsweise Ω sein. Versucht man hier das Argument zuerst zu reduzieren, so gerät man in eine Endlosschleife. Auf der anderen Seite kann das Argument ja ungenutzt bleiben - daher würde auch Ω innerhalb der Funktion niemals auftauchen und es müsste nie reduziert werden. Auf der anderen Seite garantiert das Reduzieren der äußersten, möglichen Funktion (Normal Order Reduction) das Erreichen der Normalform, sofern eine existiert [8].

Blis benutzt eine Mischform der beiden, die trotzdem Optimalität garantiert. Vor dem Einsetzen in die Funktion versucht es, den Parameter mit einer reduzierten Anzahl von Iterationen zu reduzieren. Ist der Parameter nach der angegebenen Anzahl von Iterationen noch nicht reduziert, wird er ganz normal eingesetzt. Dies verringert in der Praxis die Menge der Operationen, die verrichtet werden muss, ohne dass bestimmte Normalformen nicht mehr erreicht werden können.

3.3. Funktionsweise

Blis ist in Rust geschrieben und besteht aus drei Teilschritten - dem Kompilieren, Parsen und Reduzieren. Jeder Schritt ist dabei mit Unit-Tests versehen und weiterhin gibt es auch End-to-End-Tests mit einigen Sprachkonstrukten, die in Abschnitt 2.3. hergeleitet wurden. Zum Ausführen des Programms muss die Rust-Toolchain installiert sein. Hier die beiden Befehle, die zum Ausführen benötigt werden:

Auszuführen	Befehl
Programm	`cargo run --release`
Testsuite	`cargo test`

Beim Kompilieren werden meine syntaktischen Verbesserungen zu reinem Lambda-Kalkül kompiliert. Anschließend wird aus dem kompilierten Programmtext eine Baumstruktur geparsed. Typischerweise beinhalten diese drei Knotenpunkttypen, die jeweils mit den drei Definitionsteilen des Lamba Kalkül (Variablen, Abstraktion, Applikation) korrespondieren. Ich habe mich aus Performance- sowie Laufzeitspeichergründen für eine andere Aufteilung entschieden. Im Programm wird unterteilt in Variablen, Funktionsdefinitionen und Funktionsaufrufe. Funktionsdefinitionen sind einfach das: in ihnen wird eine Funktion definiert, die ein Argument annimmt und etwas zurückgibt. Funktionsdefinitionen beinhalten ein optionales Argument für den übergebenen Parameter, da sie entweder mit einem Parameter aufgerufen werden oder nicht. Dabei werden überflüssige Parameter den entstehenden Baum aus Lambda-Ausdrücken heruntergegeben. Diese Vorgehensweise nutzt den Fakt, dass die beiden folgenden Ausdrücke äquivalent sind:

$$\lambda x(\lambda y(M))\ N\ O \Leftrightarrow \lambda x(\lambda y(M)\ O)\ N$$

Funktionsaufrufe stehen für das Aufrufen von Funktionen, die noch nicht definiert wurden. Dafür folgendes Beispiel:

$$(\lambda f.(f\ 5))(\lambda x.x)$$

Im Funktionskörper von *f* wird *f* mit dem Parameter *5* aufgerufen. Aber ohne das Programm auszuführen, gibt es keinerlei Möglichkeiten herauszufinden, welchen Wert *f* annehmen wird, also welche Funktion es darstellt. Daher speichern Funktionsaufrufe lediglich den Namen der Funktion, sowie eine Liste mit Parametern. Es könnte ja sein, dass die Funktion mehr als einen Parameter annimmt [siehe Currying]. Beim Parsen werden außerdem die Variablennamen aufgelöst. Dabei bekommt jede Variable eine globale, eindeutige numerische ID. Dies reduziert nicht nur den Speicherbedarf, weil nicht in jeder Instanz eines Lambda-Ausdrucks ein String mitgespeichert werden muss, sondern verhindert auch das Kollidieren von Namen. Dieser Schritt ist äquivalent zur α-Umwandlung aller Variablen zu den natürlichen Zahlen.
Beim Reduzieren wird zuletzt in der “Normal-Order”-Reihenfolge eine Funktionsdefinition zum Reduzieren ausgewählt. Da es keine Namenskollisionen geben kann, ist die □-Reduktion dabei trivial.

4. Schluss

Der Lambda Kalkül wurde von Alonzo Church entwickelt und ist heute ein fester Bestandteil der Informatik. Er ist dabei ein gutes Stück theoretischer als die Turing-Maschine, sie sind aber aufgrund ihrer Turing-Vollständigkeit äquivalent. Besonders gut geeignet ist das System aufgrund seiner mathematischen Struktur für das Aufbauen von Definitionen und Abstraktionen. So kann man ausgehend von einfachsten Datenstrukturen, wie Booleans und Pairs, bereits Logikgatter und Listen erstellen. Durch das Darstellen von natürlichen Zahlen über die Church-Notation erhält der Lambda-Kalkül eine Repräsentation von Arithmetik. Mit der Definition von Kombinatoren, die Rekursion ermöglichen, erhält man eine unpraktische, wenn auch theoretisch nutzbare, funktionale Programmiersprache. Einen ähnlichen Weg haben auch die Entwickler von Lisp genommen und erstellten so eine Sprache, die einflussreicher kaum hätte sein können. In Dialekten, wie Clojure oder Racket, wird sie auch heute noch verwendet und viele ihrer Innovationen, wie zum Beispiel Garbage Collection oder Meta-Programming, oder sind heute standardmäßig in vielen Programmiersprachen enthalten.
Der von mir vorgestellte Dialekt des Lambda Kalküls zeichnet sich besonders durch eine Struktur aus, die einfaches Parsen ermöglicht. Weiterhin beinhaltet er einige syntaktische Vereinfachungen, die die Komplexität der jeweiligen Ausdrücke drastisch reduzieren, aber trotzdem trivial zum vereinfachten Syntax umgewandelt werden. Dieses Umwandeln stellt den ersten der drei Schritte dar, die der von mir vorgestellte Lambda-Kalkül-Reduzierer *Blis* durchführt. Angelehnt an das Verarbeiten von Programmiersprachen wird der Lambda-Kalkül anschließend geparsed, also in eine native Repräsentation umgewandelt. Zuletzt wird diese

native Repräsentation reduziert, bis keine Reduktionen mehr möglich sind oder das Programm entscheidet, dass dieser Ausdruck nicht reduzierbar ist. Um eine bestmögliche Reduktion zu gewährleisten, findet diese Reduktion in der *Normal Reduction Order* statt.

Blis hat einige Performance-Probleme, die ich in zukünftigen Versionen gerne beheben würde. Vor Allem wird ein großer Teil der Laufzeit mit dem Allokieren und Deallokieren von Speicherplatz verbracht. Der Reduzierer ist in der Hinsicht nicht sonderlich performant geschrieben und eine Optimierung dessen wäre ein mögliches zukünftiges Projekt. Eine weitere mögliche Weiterführung baut auf Rust's typensicheres Metaprogramming. Es ist möglich, den Lambda-Kalkül zu Compile-Time in Rust-Typen zu übersetzen. Dann könnte man auch Rust-Ausdrücke innerhalb des Lambda-Kalküls verwenden, weil am Ende alles zu Rust kompiliert wird.

5. Quellenverzeichnis

[1] The University of Birmingham, School of Computer Science (2019): “Handout 4: Simply Typed Lambda Calculus”. URL: https://www.cs.bham.ac.uk/~udr/popl/04-19-TLC.pdf [Stand 05.07.2024]

[2] Church, Alonzo (1936): “An Unsolvable Problem of Elementary Number Theory”. URL: https://ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf [Stand 05.07.2024]

[3] Ministerium für Bildung, Wissenschaft und Kultur des Landes Mecklenburg-Vorpommern (2019): “Rahmenplan Informatik 2019”. URL: https://www.bildung-mv.de/export/sites/bildungsserver/downloads/unterricht/rahmenplaene_allgemeinbildende_schulen/Informatik/RP_INFO_SEK2.pdf [Stand 05.07.2024]

[4] Encyclopedia of Mathematics: “Lambda-calculus”. URL: http://encyclopediaofmath.org/index.php?title=Lambda-calculus&oldid=50185 [Stand 05.07.2024]

[5] McCarthy, John (1979): “History of Lisp”. URL: http://jmc.stanford.edu/articles/lisp/lisp.pdf [Stand 05.07.2024]

[6] Graham, Paul (2002): “The Roots of Lisp”. URL: http://languagelog.ldc.upenn.edu/myl/ldc/llog/jmc.pdf [Stand 05.07.2024]

[7] University of Cambridge, Department of Computer Science and Technology: “Lambda calculus”. URL: https://www.cl.cam.ac.uk/teaching/1718/L28/01-lambda-notes.pdf [Stand 05.07.2024]

[8] University of Wisconsin-Madison: “Lambda Calculus (Part 1)”. URL: https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html#NOR [Stand 05.07.2024]

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen Publikationen, Vorlagen und Hilfsmitteln als die angegebenen benutzt habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt für von mir verwendete Bild- und Internetquellen. Die Arbeit ist weder von mir noch von einem anderen Schüler bisher vorgelegt worden.

Neubrandenburg, 5.7.24

Ort, Datum, Unterschrift