THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

CS5341 – Kernel-Architekturen in Programmiersprachen

Thema:
STG – Kernelsprache und Ausführungsmodell nicht-strikter funktionaler Programmiersprachen

Vorgelegt von:	Niklas Deworetzki
Matrikelnummer	5185551
Eingereicht bei Hochschulbetreuer:	Prof. Dr.-Ing. Dominikus Herzberg
Eingereicht am:	24. Januar 2022

Abstract

Diese Ausarbeitung beschäftigt sich mit den Besonderheiten der Spineless Tagless G-Machine und der dazugehörigen funktionalen Programmiersprache, welche als Kernelsprache einzuordnen ist. Anwendungsbereich der Maschine ist die Ausführung von nicht-strikten funktionalen Programmiersprachen wie etwa Haskell auf gewöhnlicher Hardware. Es wird untersucht, wie diese Maschine das Schließen der semantischen Lücke zwischen funktionalen Hochsprachen und primitiven Registermaschinen bewerkstelligt, indem eine virtuelle Maschinenimplementierung in Java erstellt wird. Die imperative Programmiersprache Java dient dabei als Abstraktion über der Maschine, die ebenfalls ein imperatives Ausführungsmodell aufweist, dabei aber mühselige Aufgaben übernimmt, die vom eigentlichen Fokus ablenken würden.

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung ist Teil der Dokumentation eines Projektes aus dem Modul *CS5341 – Kernel-Architekturen in Programmiersprachen*, welches im Wintersemester 2021/2022 stattfand.

Wie der Name es bereits verrät, liegt der Fokus der Veranstaltung auf Programmiersprachen mit einem kleinen Sprachkern, sogenannten *Kernel-Sprachen*. Diese Sprachen besitzen zumeist die Möglichkeit, sich mit eigenen Mitteln selbst zu erweitern, um so Schicht für Schicht höhere Abstraktionen aufzubauen, ohne diese explizit bei der Implementierung der Sprache zu unterstützen. Das macht diese Sprachen nicht nur für Sprachentwickler interessant, da die Implementierung eines kleinen Sprachkerns nur mit vergleichsweise geringem Aufwand verbunden ist. Auch für Anwender sind solche Sprachen interessant, da sie zumeist flexibel sind, über Bibliotheken einfach erweitert werden können und die geringe Zahl der Kernfeatures ein schnelles Erlernen fördert. Bekannte Vertreter für Kernelsprachen sind die verschiedenen Lisp-Dialekte, welche im Sprachkern lediglich Listen, Funktionen und Funktionsanwendungen bieten, während die restliche Funktionalität über Makromechanismen oder reflexive Programmierung entstehen [5, 18]. Funktionale Sprachen wie Haskell oder Scala wandeln Quellprogramme mit komplexeren Bestandteil in eine kalkülartige Kernsprache um, welche für Analysen im Compiler und die Ausführung herangezogen wird [6, 12]. Bei den objektorientierten Programmiersprachen ist Smalltalk als wichtiger Vertreter zu nennen [4]. Hier werden alle Sprachkonstrukte als Objekte dargestellt, welche sich Nachrichten senden können.

Während in den seminaristischen Vorlesungsstunden der Veranstaltung der Fokus zumeist auf den Mechanismen liegt, die in solchen Programmiersprachen verwendet werden, liegt der Fokus dieses Projektes in der Auseinandersetzung mit einem Programmiersprachenkern und besonders auf dem Ausführungsmodell, das diesem zugrunde liegt. Die sogenannte *STG-Sprache* wird verwendet, um eine Ausführungsumgebung für Haskell bereitzustellen, weswegen die Sprache nicht auf hoher Flexibilität und Erweiterbarkeit sondern vielmehr auf Maschinennähe und Kontrolle der ausgeführten Operationen zur Laufzeit basiert. Die Besonderheiten dieser

Programmiersprache wurden im Rahmen des Projektes untersucht und eine Implementierung der dazugehörigen Maschine in der imperativen Programmiersprache Java erstellt.

Diese Ausarbeitung zum Projekt ist in vier weitere Teile gegliedert, welche die verschiedenen Abschnitte des Projektes widerspiegeln.

- Kapitel 2 befasst sich mit den Grundlagen hinter der STG-Sprache und der dazugehörigen Maschine.
- Kapitel 3 beschreibt die für die Sprache definierten Sprachkonstrukte und deren Bedeutung zur Ausführungszeit.
- Kapitel 4 assoziiert die einzelnen Sprachkonstrukte mit der zugehörigen Semantik sowie der Implementierung dieser für eine virtuelle Maschine in Java.
- Kapitel 5 fasst die Ergebnisse des Projektes zusammen und reflektiert diese.

2 Grundlagen

Die Spineless Tagless G-Machine (STG) beschreibt eine abstrakte Maschine, sowie eine kleine Programmiersprache zur Programmierung ebendieser Maschine, die als STG-Sprache bezeichnet wird [14]. Der vorwiegende Verwendungszweck liegt dabei in der Übersetzung und Ausführung von nicht-strikten funktionalen Programmiersprachen. In solchen Programmiersprachen werden Ausdrücke verzögert und nur bei Bedarf ausgewertet. Man spricht auch von der sogenannten *Bedarfsauswertung* oder aus dem Englischen *lazy Evaluation*.

Dieser Ausführungsmodus kommt mit einer Reihe an großen Herausforderungen. Zur Laufzeit muss zwischen ausstehenden oder unterbrochenen Auswertungen (sogenannten *Thunks*) und bereits berechneten Werten unterschieden werden. Gleichzeitig soll überflüssige Arbeit vermieden werden, indem Ausdrücke nur so oft wie nötig ausgewertet werden, um anschließend deren Wert für den Falle erneuter Auswertung zu speichern. Zudem ist es in funktionalen Sprachen häufig der Fall, dass Ausdrücke nicht nur einfache Werte sondern auch Funktionen berechnen, wofür sogenannte *Closures* verwendet werden, die Funktionsimplementierung und die für die Funktion sichtbaren Variablen Speichern. Selbstverständlich soll eine Ausführungsumgebung, die all diese Anforderungen unterstützt, auch noch möglichst effizient sein und mit möglichst wenig Speicherbedarf und Rechenzeit auskommen.

Die STG-Maschine verspricht, als abstrakte Maschine diesen Anforderungen nachzukommen und wird seit den 1990er Jahren als wesentlicher Bestandteil in der Implementierung und Übersetzung von Haskell verwendet [2]. Im Ergebnis ist Performance von übersetzten Haskell Programmen häufig vergleichbar mit C [13].

2.1 Graphenreduktion als Ausführungsmodell

Die Verwendung einer abstrakten Maschine zur Ausführung einer Programmiersprache ist keine Besonderheit. Die ursprünglich für Java entwickelte Java Virtual Machine (JVM) beschreibt eine abstrakte Stackmaschine, die entweder im Rahmen einer virtuellen Maschinenimplementierung ausgeführt wird, oder deren Seman-

tik in Maschinencode für eine reale Maschine übersetzt wird [10]. Um die .NET Plattform herum entstand die Common Language Infrastructure (CLI), welche standardisiert eine objektorientierte abstrakte Stackmaschine beschreibt. In beiden Fällen soll die Verwendung einer abstrakten Maschine über die Tatsächliche Ausführungsumgebung abstrahieren, um so Programme plattform- und hardwareunabhängig ausführen zu können [7]. Eine ähnliche Architektur bietet LLVM mit einer abstrakten Registermaschine [19]. Hier liegt der Fokus jedoch auf der Optimierung und Übersetzung von Programmen für diese abstrakte Maschine in realen Maschinencode.

All diese Modelle sind nah an dem Modell der handelsüblichen Computer, welche auch als sogenannte Sequentielle Maschinen Einzug in die theoretische Welt der Berechenbarkeit gehalten haben [17]. Die Grundannahmen sind hier, dass Rechenschritte einzelne gleichwertige Instruktionen darstellen, die nacheinander ausgeführt werden und jeweils eine Menge an Registern oder Speicherzellen beeinflussen.

Die STG-Maschine wählt hier einen anderen Ansatz, der näher am Berechnungsvorgehen des Lambda-Kalküls ist: Hier liegt das Programm als Datenstruktur vor, in der Ausdrücke als Knoten mit Kanten zu den verwendeten Teilausdrücken vorkommen. Reduktionsregeln geben vor, wie schrittweise diese Datenstruktur reduziert werden kann, bis der Ausdruck schließlich eine sogenannte Normalform erreicht – eine einheitliche Form des Ausdrucks, die dessen ausgewerteten Wert repräsentiert [21].

Der Teil *G-Maschine* im Namen der STG-Maschine steht dabei für *Graphenreduktionsmaschine*. Das auszuwertende Programm liegt also als Graph vor, der schrittweise reduziert wird. Die Darstellung als Graph ermöglicht dabei eine genauere Beschreibung der Laufzeitsemantik eines Programmes, als es durch einen Baum möglich ist. Darum wird diese Darstellung der klassischen Repräsentation als Syntaxbaum vorgezogen. Abbildung 2.1 zeigt beispielhaft einen solchen Graphen für den Ausdruck $\texttt{let}\ x\ =\ \frac{2}{y}\ \texttt{in}\ x + x$. Die Darstellung als Graph verdeutlicht hier, dass die beiden Vorkommnisse von x auf denselben Wert verweisen. Wird nun für y der Wert 1 bekannt, können nacheinander Reduktionsregeln angewandt werden, um den Graphen wie in Abbildung 2.1 auf einen einzelnen atomaren Wert zu reduzieren.

2.2 Besonderheiten der STG

Die STG-Maschine definiert einige Erweiterungen für dieses Modell der Graphenreduktion. Mit Hilfe dieser wird lazy Evaluation möglich und die effiziente Ausführung auf gewöhnlicher Hardware unterstützt.

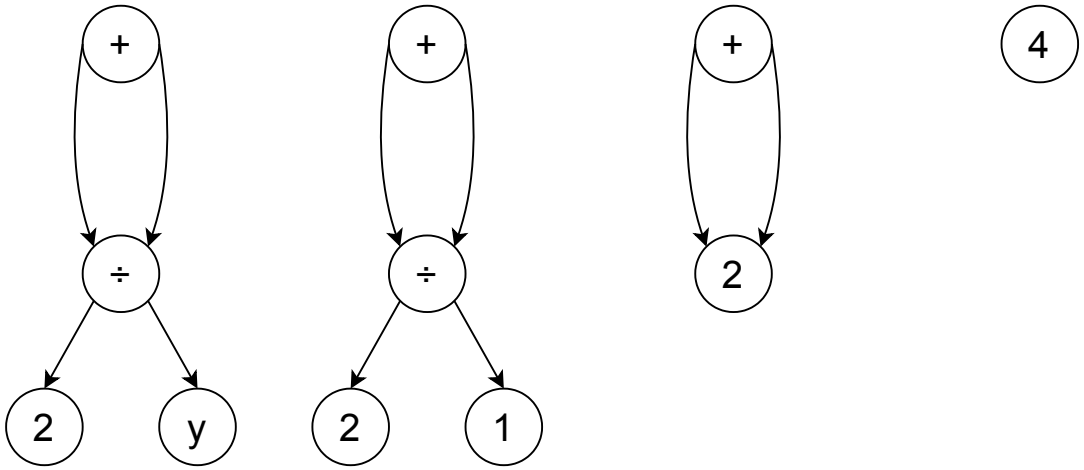

Abbildung 2.1: Schrittweise Redutkion des Ausdrucks `let` $x = \frac{2}{y}$ `in` $x + x$

Während die Darstellung eines Programms und dessen Ausdrücke als Knoten in einem Graph über die verschiedenen Ausprägungen der Laufzeitwerte abstrahiert und so im grundlegenden Modell die Unterscheidung zwischen Thunks, Werten und Closures überflüssig macht, müssen die durchgeführten Reduktionsregeln weiter eingeschränkt werden, um lazy Evaluation zu ermöglichen. Dem Ansatz aus Abbildung 2.1 entsprechend würde der gesamte Graph reduziert, was der Idee der lazy Evaluation widerspricht.

Die Lösung ist das Definieren einer Normalform, die nicht zwingend einen atomaren Wert darstellt. Die Anwendung von Reduktionsregeln auf einen Teilgraphen in Normalform verändert diesen nicht mehr. So wird strikte Auswertung unterbunden. Im Rahmen der STG wird die Weak Head Normal Form (WHNF) verwendet [16]. In WHNF muss lediglich der Kopf eines Graphen in Normalform vorliegen, um ihn als ausgewertet zu betrachten. Dies ist der Fall, wenn der Wurzelknoten des Graphen einen Konstruktor, einen atomaren Wert oder eine eingebaute Funktionsanwendung beschreibt. Der Rest des Graphen darf dabei unausgewertet vorliegen, wodurch die WHNF zu einer schwachen Normalform wird. Am konkreten Beispiel einer verketteten Liste bedeutet dies, dass lediglich bekannt sein muss, ob der Kopf der Liste den Listenkonstruktor oder die leere Liste darstellt. Ob die Liste abgesehen vom Kopf weitere Elemente enthält, wie lang diese Liste ist, oder wie das vorderste Listenelement aussieht, ist dabei für die WHNF irrelevant.

Problematisch wird trotz der WHNF dennoch die Auswertung von unendlichen oder zyklischen Graphen. Gerade unendlich große Graphen erweisen sich als Herausforderung, wenn man den naiven Ansatz verfolgt und den gesamten Programmgraphen als Datenstruktur im Speicher hat. Diesen Problemfall behandelt das *S*

im Namen der STG, welches für *Spineless* steht. Die sogenannte Spine bezeichnet dabei die Datenstruktur, welche im Hauptspeicher einer Maschine den Graphen enthält. Bei der STG existiert diese Datenstruktur nicht explizit. Stattdessen werden Zeiger auf berechnete Werte oder Codeblöcke verwendet, die den Graphen repräsentieren. Als Resultat wird während der Laufzeit nicht der gesamte Graph im Speicher gehalten, sondern nur die Teile, die für die aktuelle Auswertung relevant sind.

Die letzte Erweiterung der STG wird durch den Buchstaben T im Namen beschrieben, welches für *Tagless* steht. Obwohl die Knoten im Modell der Graphenausführung Closures, Thunks und Werte vereinheitlichen, kann es nötig sein, während der Auswertung eines Ausdrucks, zwischen diesen Ausprägungen zu unterscheiden. Ist der zu reduzierende Teil des Graphen ein Thunk, so muss die von diesem dargestellte unterbrochene Berechnung fortgesetzt werden. Liegt ein bereits ausgewerteter Wert vor, so kann dieser direkt verwendet werden.

Die Unterscheidung der Ausprägungen geschieht in anderen Graphenmaschinen durch sogenannte Tags, welche im Knoten gespeichert diesen markieren und dessen Ausprägung bestimmen [8, 1, 9]. Die STG verwendet eine einheitliche Darstellung, in der alle Ausprägungen als Closure dargestellt werden. Neben den freien Variablen, die sowohl bei herkömmlichen Closures als auch bei Thunks eingefangen und gespeichert werden müssen, enthält die Darstellung auch einen Zeiger auf einen Codeblock anstelle eines Tags. So wird es überflüssig, die korrekte Aktion zur Auswertung der Closure anhand des Tags zu bestimmen. Der Zeiger verweist stattdessen auf den Code, der die gewünschte Aktion beschreibt und ein einfacher Sprung genügt, um diese auszuführen. Closures können so Argumente vom Stack konsumieren und die Auswertung des Rumpfs beginnen, für Thunks wird die unterbrochene Ausführung angestoßen und für bereits ausgewertete Werte der jeweilige Wert zurückgegeben.

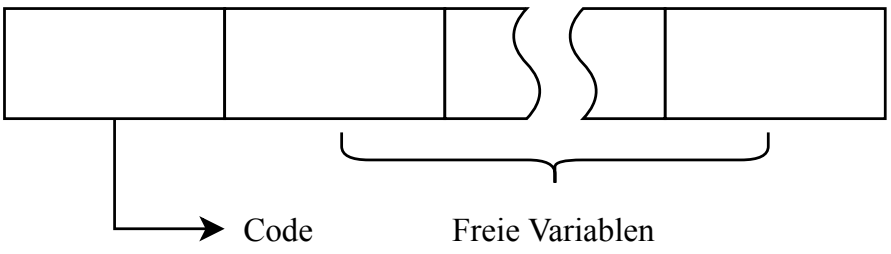

Abbildung 2.2: Aufbau einer Closure im STG-Modell

3 Die STG-Sprache

Die STG-Sprache ist eine kleine funktionale Programmiersprache, die eng mit der Semantik der STG-Maschine verknüpft ist. Zudem dient sie als kleiner Sprachkern für Haskell und wird dort im Übersetzungs- und Ausführungsprozess verwendet.

Im Vergleich zu anderen Maschinensprachen fällt auf, dass die STG-Sprache eine funktionale Programmiersprache ist. Anstelle der Beschreibung einzelner Anweisungen oder Instruktionen, die den Zustand der Maschine ändern, werden deklarativ Funktionen beschrieben, deren Zusammenspiel einen Graphen bildet. Da die Reduktion dieses Graphen der Ausführung der STG-Maschine entspricht, gibt es einige Unterschiede zu funktionalen Kern- oder Hochsprachen. Anstelle viele, einfache Abstraktionen zu schaffen, ist eine genaue Kontrolle über das Verhalten der zugrundeliegenden Maschine gewünscht.

Sowohl die Nähe zur Maschinensemantik als auch die Kontrolle der selben wird bei Betrachtung der verschiedenen Sprachkonstrukte deutlich, die im Folgenden vorgestellt werden.

3.1 Lambdaformen

Lambdaformen entsprechen den klassischen Lambdaausdrücken, wie sie aus nahezu allen funktionalen Sprachen bekannt sind. Zusätzlich zu der Liste an Variablen, an welche die Argumente der anonymen Funktion gebunden werden, und dem Rumpf der Funktion existieren einige Besonderheiten.

Eine zusätzliche Liste an Variablen bei der Definition einer Lambdaform beschreibt die freien Variablen, die beim Erstellen einer Closure gespeichert werden müssen. Freie Variablen sind Variablen, die in einem Ausdruck verwendet werden, jedoch nicht innerhalb dieses Ausdrucks definiert werden. Sie entstehen also durch eine umschließende Definition, und müssen als Kontext der Funktion gespeichert werden. Da dies Speicherplatz benötigt und somit die Ausführung der STG-Maschine beeinflusst, werden die freien Variablen explizit angegeben.

Program	$prog$	$\rightarrow$ $binds$	
Bindings	$binds$	$\rightarrow$ var_1 `=` lf_1 `;` ... `;` var_n `=` lf_n	$n \geq 1$
Lambda-forms	lf	$\rightarrow$ $vars_f$ `\`π $vars_a$ `->` $expr$	
Update flag	π	$\rightarrow$ `u`	Updatable
		| `n`	Not updatable
Expression	$expr$	$\rightarrow$ `let` $binds$ `in` $expr$	Local definition
		| `letrec` $binds$ `in` $expr$	Local recursion
		| `case` $expr$ `of` $alts$	Case expression
		| var $atoms$	Application
		| $constr$ $atoms$	Saturated constructor
		| $prim$ $atoms$	Saturated built-in op
		| $literal$	
Alternatives	$alts$	$\rightarrow$ $aalt_1$ `;` ... `;` $aalt_n$ `;` $default$	$n \geq 0$ (Algebraic)
		| $palt_1$ `;` ... `;` $palt_n$ `;` $default$	$n \geq 0$ (Primitive)
Algebraic alt	$aalt$	$\rightarrow$ $constr$ $vars$ `->` $expr$	
Primitive alt	$palt$	$\rightarrow$ $literal$ `->` $expr$	
Default alt	$default$	$\rightarrow$ var `->` $expr$	
		| `default` `->` $expr$	
Literals	$literal$	$\rightarrow$ `0#` | `1#` | ...	Primitive integers
		| ...	
Primitive ops	$prim$	$\rightarrow$ `+#` | `-#` | `*#` | `/#`	Primitive integer ops
		| ...	
Variable lists	$vars$	$\rightarrow$ `{`var_1`,` ... `,` var_n`}`	$n \geq 0$
Atom lists	$atoms$	$\rightarrow$ `{`$atom_1$`,` ... `,` $atom_n$`}`	$n \geq 0$
	$atom$	$\rightarrow$ var | $literal$	

Abbildung 3.1: Grammatik der STG-Sprache aus [14]

Weiterhin fällt auf, dass eine *Update Flag* als Teil der Lambdaform angegeben wird. Ist diese Flagge auf `u` gesetzt, wird die Closure im Speicher nach der Auswertung durch den Ergebniswert ersetzt. Somit wird eine erneute Auswertung unterbunden, die Performance erhöht und keine zusätzliche Berechnungen durchgeführt. In anderen Fällen ist diese Ersetzung nicht erwünscht. Wird ein Wert beispielsweise nur einmal berechnet, muss keine Ersetzung stattfinden und der dafür benötigte Aufwand kann eingespart werden. Ähnlich ist es, wenn die Lambdaform eine Funktion beschreibt, oder der Ausdruck im Rumpf bereits in der WHNF ist (siehe [14, Chap. 4.2]).

3.2 Let-Bindungen

Bindungsausdrücke binden (potentiell mehrere) Lambdaformen an Bezeichner. Hierbei wird zwischen Bindungsausdrücken mit dem Schlüsselwort `let` und rekursiven Bindungsausdrücken mit dem Schlüsselwort `letrec` unterschieden. Bei Letzterem dürfen die Definitionen des Ausdrucks gegenseitige Querbezüge besitzen. Bei der ersten Variante ist dies nicht erlaubt.

Als Besonderheit fällt auf, dass lediglich Lambdaformen gebunden werden können. Dies reflektiert die lazy Evaluation, da so nicht ein Ausdruck selbst oder dessen Wert gebunden wird.

Hinsichtlich der Maschinensemantik beschreiben Bindungsausdrücke immer Speicherallokation. Für jede gebundene Lambdaform wird eine Closure auf dem Heap angelegt und die Referenz auf diese für die gebundene Variable verwendet.

3.3 Anwendungen

Die STG-Maschine kennt Anwendungen von Funktionen, Konstruktoren und primitiven Operationen. Im Vergleich zu Haskell gilt hier die Einschränkungen, dass die Anwendungen von Konstruktoren und primitiven Operationen immer alle Argumente angegeben sein müssen. Eine η-Reduktion, wie sie in Haskell üblich und idiomatisch ist, wird hier nicht durchgeführt. So ist sichergestellt, dass immer genügend Argumente auf dem Stack liegen, wenn diese Konstrukte ausgewertet werden, wodurch das Erreichen der WHNF bei der Auswertung von Konstruktoranwendungen und Aufrufe von primitiven Operationen sichergestellt wird.

Als Einschränkungen gilt für diese Ausdrücke, dass die übergebenen Argumente bei einer Anwendung immer atomar sein müssen; lediglich primitive Konstanten und Variablen sind erlaubt. Dies reduziert die Komplexität der Maschine und erzwingt,

dass komplexe Argumente, die ausgewertet werden müssten, explizit als Closure auf dem Heap abgelegt werden, bevor diese als Argument übergeben werden.

Soll eine einzelne Variable als Ausdruck verwendet werden, so muss die Variable als Funktion auf eine leere Parameterliste angewandt werden. Bei der Auswertung dieses Ausdrucks wird dann zum Rumpf der Closure gesprungen, die an diese Variable gebunden wird. Diese etwas umständliche Einschränkung bildet syntaktisch direkt das Ausführungsmodell der STG-Maschine ab, in der jeder Ausdruck verzögert ausgewertet wird.

3.4 Fallunterscheidungen

Fallunterscheidungen bestehen aus einem untersuchten Ausdruck und einer Reihe an Fällen, die jeweils ein Muster definieren. Der erste Fall, dessen Muster zu dem untersuchten Ausdruck passt, wird dabei als Ergebnis der Fallunterscheidung ausgewählt. Ist kein passendes Muster gegeben, so tritt ein Laufzeitfehler auf.

Fallunterscheidungen sind einstufig; sie können nur anhand des äußersten Konstruktors oder einer primitiven Konstante getroffen werden. Zudem darf ein Standardfall definiert werden, der immer akzeptiert und optional den gesamten Ausdruck an einen Namen bindet. In der Praxis stellt diese Einschränkung kein Problem dar, da mehrstufige Fallunterscheidungen in einstufige übersetzt werden können [20].

Die Besonderheit in der Semantik von Fallunterscheidungen ist, dass der untersuchte Ausdruck ausgewertet wird. Somit bieten Fallunterscheidungen die einzige Möglichkeit in der STG, die Auswertung eines Ausdrucks zu erzwingen. Durch die Auswertung des untersuchten Ausdrucks in WHNF wird sichergestellt, dass der korrekte Fall ausgewählt wird, da nach der Auswertung der passende Konstruktor in ausgewerteter Form vorliegt. Gleichzeitig wird dabei sichergestellt, dass nicht zu viel ausgewertet wird und die Laziness erhalten bleibt.

3.5 Primitive Werte und Arithmetik

In reinen nicht-strikten Programmiersprachen werden Zahlenwerte und arithmetische Operationen – wie alle weiteren Operationen auch – als Closures oder Thunks dargestellt, welche die Berechnungen enthalten. Folglich sind arithmetische Operationen mit hohen Laufzeitkosten verbunden. Eine einfache Addition zweier Zahlen etwa, erzwingt das Auswerten der Operanden, das Entpacken der ausgewerteten Zahlenwerte, die eigentliche Durchführung der Addition, das Anlegen einer neuen Closure für den Ergebniswert und das anschließende Ablegen des Ergebnisses.

Die STG bietet durch primitive Werte eine Möglichkeit, verfügbare Maschinendatentypen und Maschineninstruktionen einer echten Hardware direkt in der abstrakte Maschine abzubilden. So existiert beispielsweise der primitive Datentyp `Int#`, welcher ganzzahlige Maschinenworte darstellt. Arithmetische Operationen auf diesen primitiven Werten sind beispielsweise als `+#`, `-#` verfügbar.

Ganz dem Sinne einer Kernelsprache entsprechend, kann ein verzögert ausgewerteter Zahlentyp um diese primitiven Datentypen herum implementiert werden.

```
data Int = MkInt Int#
```

Eine solche Definition beschreibt einen algebraischen Datentypen `Int` mit einem einzelnen Konstruktor `MkInt`, der eine primitive Ganzzahl akzeptiert.

Arithmetische Operationen, welche die verzögerte Auswertung unterstützen, können auf ähnliche Weise definiert werden, indem Fallunterscheidungen zum Auswerten und Auspacken der Operanden verwendet und das Ergebnis in den Konstruktor des Zahlentyps gepackt wird. Ein Ausdruck `(e1 + e2)` in einer höheren Sprache wie Haskell, könnte wie folgt umgeschrieben werden, um verzögerte Auswertung zu unterstützen:

```
case e1 of
MkInt x# -> case e2 of
              MkInt y# -> case (+# x# y#) of
                            r# -> MkInt r#
```

Die Argumente werden genau wie die primitive Addition durch eine Fallunterscheidung ausgewertet. Das Ergebnis der Addition wird an den Bezeichner `r#` gebunden und an den Konstruktor des Zahlentyps übergeben. Im gezeigten Programmausschnitt wird die Konvention angewandt, Bezeichner für primitive Variablen mit dem Suffix `#` zu versehen, wie es bei den primitiven arithmetischen Operationen ebenfalls der Fall ist.

4 Implementierung

Das Kernziel des Projektes ist es, die formale Notation der operationellen Semantik in eine ausführbare Implementierung umzuwandeln. Die Wahl der Implementierungssprache fällt hier auf Java, eine imperative und objektorientierte Programmiersprache. Als Notation zur Beschreibung der Semantik werden in [14] Zustandsübergänge gewählt, welche in Schreibweise einer Funktion auf bestimmte Muster des Maschinenzustands greifen und auf den nächsten Zustand der Ausführung abbilden. Diese Notation kann nicht ohne Weiteres in die imperative Programmiersprache Java übersetzt werden. Doch genau hier liegt der Reiz des Projektes, da die STG genau für diesen Zweck gedacht ist, die semantische Lücke zwischen verzögert ausgewerteten funktionalen Hochsprachen und dem hardwarenahen imperativen Ausführungsmodell herkömmlicher Hardware zu schließen. Zwar wird Java kaum als hardwarenahe Programmiersprache verwendet, die imperativen Eigenschaften sind jedoch vorhanden und Features wie Objektorientierung und automatische Speicherverwaltung unterstützen die zielgerichtete Implementierung.

4.1 Der Maschinenzustand

Der Zustand der STG-Maschine besteht aus insgesamt fünf Komponenten, die in Java abgebildet werden:

1. Der *Maschinencode*, welcher eine feste Anzahl an Ausprägungen annehmen kann,
2. der *Argument-Stack*, welcher Werte enthält,
3. der *Return-Stack*, welcher *Continuations* speichert,
4. der *Update-Stack*, welcher Informationen zur Ersetzung von Closures speichert, und
5. der *Heap*, welcher die Closures speichert.

Zudem existiert eine statische, globale Umgebung, welche die Adressen für alle Closures liefert, die auf oberster Programmebene definiert sind. Diese Umgebung ist während der gesamten Laufzeit unverändert.

Die drei verschiedenen Stacks werden als `Deque` aus dem Paket `java.util` dargestellt. Für die Elemente, die diese Stacks speichern, werden verschiedene Klassen angelegt, um die einzelnen Ausprägungen darzustellen. Continuations und die Informationen zur Ersetzung von Closures (genannt Update-Rahmen) werden jeweils als Records in Java dargestellt.

Für Werte, die auf dem Argument-Stack liegen, existieren zwei Ausprägungen: Ganzzahlen und Adressen. Es wird für beide jeweils ein Java-Record erstellt, welcher einer gemeinsame Schnittstelle implementiert.

```
public sealed interface Value {
  record Address(int address) implements Value {}

  record Int(int value) implements Value {}
}
```

Listing 4.1: Darstellung von Werten in Java

Ähnlich wird bei den Ausprägungen des Maschinencodes vorgegangen. Da diese den Zustand der Maschine maßgeblich bestimmen, wird eine Schnittstelle `State` definiert, die von den verschiedenen Ausprägungen implementiert wird. Diese Schnittstelle definiert dabei eine Methode `State transfer(Machine)`, die den nächsten Maschinenzustand zurückgibt. Auf diese Weise wird das *Tagless*-Verhalten der STG simuliert, indem keine expliziten Fallunterscheidungen für den aktuellen Ausführungszustand getroffen werden müssen. Stattdessen wird polymorph zur richtigen Aktion für den aktuellen Ausführungszustand gesprungen. Die vier Ausprägungen, die der Maschinencode annehmen kann, sind: *Eval*, *Enter*, *Return Constructor* und *Return Integer*.

```
public sealed interface State {
    State transfer(final Machine machine);
}
```

Listing 4.2: Schnittstelle für die Ausprägungen des Maschinencodes

4.2 Auswertung von Atomen

Um Atome auszuwerten, wird eine Funktion `val` definiert, die auf die lokale Umgebung ρ und die globale Umgebung σ zugreift [14]. Diese Funktion wertet eine Konstante zu ihrem Zahlenwert aus und sucht den Wert einer Variablen in der umliegenden Umgebung. Dabei wird die lokale Umgebung der globalen Vorgezogen. Die Implementierung dieser Funktion in Java wird `value` genannt.

$$val(\rho, \sigma, x) = \begin{cases} Int\ x & \text{falls } x \text{ eine Konstante} \\ \rho\ x & \text{falls } x\ \in\ dom(\rho) \\ \sigma\ x & \text{andernfalls} \end{cases}$$

Abbildung 4.1: Funktion zur Auswertung von Atomen

4.3 Startzustand

Der Startzustand beginnt mit der Auswertung der `main` Funktion, die implizit durch die Maschine aufgerufen wird.

```
private State state = new Eval(
 new FunctionApplication(new Variable("main"), emptyList()),
 emptyMap());
```

Die etwas verbose Darstellung in Java beschreibt detailliert den Startzustand: Es wird mit der Auswertung einer Funktionsanwendung begonnen. Die angewandte Funktion ist dabei durch die Variable `main` definiert und besitzt eine leere Parameterliste. Zudem ist die Umgebung, in der die Auswertung stattfindet, leer.

Die verschiedenen benötigten Stacks werden ebenfalls leer initialisiert.

```
private Deque<Value> argumentStack = emptyStack();
private Deque<Continuation> returnStack = emptyStack();
private Deque<UpdateFrame> updateStack = emptyStack();

private final Heap heap = new Heap();
private final Map<Variable, Value> globalEnvironment =
   allocateAll(heap, program.bindings, emptyMap(), true);
```

Letztlich werden Heap und die globale Umgebung auch mit den Standardwerten

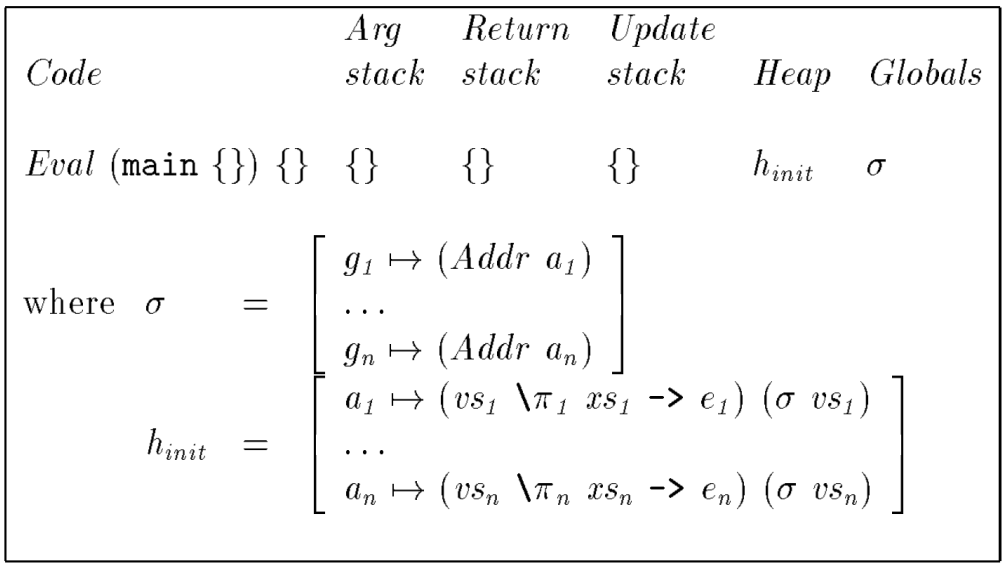

Abbildung 4.2: Startzustand der Maschine

initialisiert. Die Implementierung der Funktion `allocateAll`, welche verwendet wird, um alle globalen Namensbindungen in die globale Umgebung einzutragen und gleichzeitig auf dem Heap anzulegen, wird in Abschnitt 4.5 näher gezeigt.

4.4 Funktionsanwendungen

Die Semantik zur Auswertung von Funktionanwendungen wird in [14] durch Regel 1 und Regel 2 gegeben. Vor dem Pfeil in dieser Notation steht ein Muster, das die Komponenten des Maschinenzustands gegeben durch Reihenfolge listet: *Maschinencode, Argument-Stack, Return-Stack, Update-Stack*, *Heap* und die globale Umgebung σ. Nach dem Pfeil steht der folgende Maschinenzustand.

In der Implementierung in Java wird der Zustand *Eval* durch das Visitor-Muster abgebildet [3]. Es wird also für jeden möglichen Ausdruck, der ausgewertet werden kann eine `State visit(`*`Ausdruck`*`)` Methode implementiert, die eine konkrete Instanz von *`Ausdruck`* auswertet und den Zustand für den nächsten Auswertungsschritt zurückliefert.

Listing 4.3 zeigt die Auswertung von Funktionsanwendungen. Zunächst wird die Funktion mittels `value` ausgewertet wird und prüft, ob eine Adresse als Ergebniswert vorliegt. Ist dies der Fall, werden die Argumente ausgewertet, mit dem Argumentstack konkateniert und ein neuer Zustand zurückgegeben, der die Closure betreten und den darin enthaltenen Code ausführen soll.

```
public State visit(FunctionApplication app) {
  final Value function = value(rho, sigma, app.function);
  if (function instanceof Address a) {
    List<Value> args = values(rho, sigma, app.arguments);
    concat(args, argumentStack);
    return new Enter(a.address());
  }
}
```

Listing 4.3: Auswertung von Funktionsanwendungen

$$
\begin{array}{ll}
 & \mathit{Eval}\ (f\ \ xs)\ \rho \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad as\ \ rs\ \ us\ \ h\ \ \sigma \\
(1) & \text{such that}\ \ \mathit{val}\ \rho\ \sigma\ f = \mathit{Addr}\ a \\
 & \Longrightarrow \qquad \mathit{Enter}\ a \qquad\qquad (\mathit{val}\ \rho\ \sigma\ xs) +\!\!+ as\ \ rs\ \ us\ \ h\ \ \sigma
\end{array}
$$

Abbildung 4.3: Auswertung von Funktionsanwendungen

Die zweite relevante Regel für die Auswertung von Funktionsanwendungen beschreibt das Betreten in eine Closure. Hierbei wird die Closure als Adresse dargestellt und der entsprechende Wert vom Heap geholt. Wie Abbildung 4.4 zeigt, wird für die Auswertung eine lokale Umgebung zusammengestellt. Diese bildet die Namen der freien Variablen auf die in der Closure gespeicherten Werte und die definierten Parameter auf die Argumente auf dem Argument-Stack ab. Dabei werden die Werte vom Argument-Stack konsumiert und der Maschinenzustand geht anschließend zur Auswertung des Closure-Rumpfes über.

```
public record Enter(int address) implements State {
  public State transfer(final Machine machine) {
    final Closure closure = machine.heap.get(address);

    if (machine.argumentStack.size()
         >= closure.code().parameters.size()) {
      final var rho = mkLocalEnv(...);
      return new Eval(closure.body(), rho);
    }
  }
}
```

Listing 4.4: Implementierung des Betretens einer Closure

$$
\begin{array}{ll}
 & \mathit{Enter}\ a \quad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \mathit{xs}\ \texttt{->}\ e)\ \mathit{ws}_f]\ \sigma \\
 & \\
\text{such that} & \mathit{length}(\mathit{as}) \geq \mathit{length}(\mathit{xs}) \\
 & \\
(2)\ \Longrightarrow & \mathit{Eval}\ e\ \rho\ \mathit{as}'\ \mathit{rs}\ \mathit{us}\ h \qquad \sigma \\
 & \\
\text{where} & \mathit{ws}_a +\!\!+ \mathit{as}' = \mathit{as} \\
 & \mathit{length}(\mathit{ws}_a) = \mathit{length}(\mathit{xs}) \\
 & \rho = [\mathit{vs} \mapsto \mathit{ws}_f,\ \mathit{xs} \mapsto \mathit{ws}_a]
\end{array}
$$

Abbildung 4.4: Betreten einer Closure

4.5 Namensbindungen

In der STG-Sprache sind Namensbindungen für Speicherallokation zuständig. Folglich ist dies ein großer Aufgabenteil bei der Auswertung von Namensbindungen. Zusätzlich muss den innere Ausdruck der Namensbindung ausgewertet werden und die Umgebung für diesen aufgebaut.

In Abbildung 4.5 wird Regel 3 aus [14] dargestellt. Im dargestellten Fall, der die Auswertung von nicht-rekursiven Namensbindungen mit `let` beschreibt, ist es möglich, die drei Aufgabenteile strikt zu trennen. Für rekursive Namensbindungen mit `letrec` ist dies nicht möglich, da dann $\rho_{rhs} = \rho'$ gilt und das Aufbauen

der Umgebung mit dem Anlegen der Closures auf dem Heap verzahnt ist. Um die durch den Ausdruck definierten Closures zu erstellen, müssen die Adressen aller im Ausdruck definierter Closures bekannt sein. Damit eine Closure auf den Heap gelegt werden kann, muss diese jedoch erstellt sein.

$$
(3)\quad
\begin{array}{l}
\quad Eval \begin{pmatrix} \texttt{let} & x_1 \texttt{ = } vs_1 \texttt{ \textbackslash}\pi_1 \ xs_1 \texttt{ -> } e_1 \\ & \cdots \\ & x_n \texttt{ = } vs_n \texttt{ \textbackslash}\pi_n \ xs_n \texttt{ -> } e_n \\ \texttt{in } e & \end{pmatrix} \ \rho \ as \ rs \ us \ h \ \sigma \\
\Longrightarrow \quad Eval \ e \ \rho' \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad as \ rs \ us \ h' \ \sigma \\
\text{where} \quad \rho' \ = \ \rho[x_1 \mapsto Addr \ a_1, \ldots, x_n \mapsto Addr \ a_n] \\
\qquad\qquad h' \ = \ h \begin{bmatrix} a_1 \mapsto (vs_1 \texttt{ \textbackslash}\pi_1 \ xs_1 \texttt{ -> } e_1) \ (\rho_{rhs} \ vs_1) \\ \cdots \\ a_n \mapsto (vs_n \texttt{ \textbackslash}\pi_n \ xs_n \texttt{ -> } e_n) \ (\rho_{rhs} \ vs_n) \end{bmatrix} \\
\qquad\qquad \rho_{rhs} \ = \ \rho
\end{array}
$$

Abbildung 4.5: Regel zur Auswertung von Namensbindungen

Als Lösung für diese gegenseitige Abhängigkeit wird die Funktion `allocateAll` aus Listing 4.5 verwendet. Die Auswertung von Namensbindungen ist mit Hilfe der Funktion `allocateAll` trivial:

```
public State visit(LetBinding let) {
  final var env = allocateAll(heap, let.bindings, rho, let.
   isRecursive);
  return new Eval(let.expression, env);
}
```

Die Funktion `allocateAll` akzeptiert den Heap, eine Liste aus Namensbindungen und eine Umgebung, in der die Namensbindungen stattfinden als Parameter. Zusätzlich wird angegeben, ob die Namensbindung rekursiv erfolgen soll. Als Effekt legt diese Funktion Closures für die zu bindenen Namen auf dem Heap an und gibt anschließend eine Umgebung zurück, in der die gebundenen Namen sichtbar sind. Die gewählte Strategie hierfür ist, für jeden Namen eine Adresse auf dem Heap zu reservieren. Die jeweilige Adresse genügt, damit die gebundenen Namen aufeinander verweisen können. Anschließend wird die Eigenschaft der STG ausgenutzt, Closures auf dem Heap aktualisieren zu können (siehe Abschnitt 4.7). Die nun

erstellten Closures werden an die jeweils reservierte Adresse im Heap platziert. In der Implementierung taucht ein Kombinator `combineWith` auf, welcher zwei iterierbare Strukturen akzeptiert sowie eine Funktion, die paarweise auf die Elemente der beiden Datenstrukturen angewandt wird. Durch ihn werden die reservierten Adressen mit den Namen und erstellten Closures verbunden.

```
private static Map<Variable, Value> allocateAll(
  final Heap heap, final List<Bind> bindings,
  final Map<Variable, Value> outerEnvironment,
  final boolean isRecursive) {
  // Lokale Umgebung wird erstellt mit allen sichtbaren
   Namen der umliegenden Umgebung.
  final var localEnvironment =
    new HashMap<>(outerEnvironment);

  // Reserviere die Adressen aller Definitionen und trage
   sie in der Umgebung ein.
  final Iterable<Integer> addresses = heap.reserveMany(
   bindings.size());
  combineWith(addresses, bindings, (addr, bind) -> {
    localEnvironment.put(bind.variable, new Address(addr))
  });

  final var rhsEnvironment = (isRecursive) ?
   localEnvironment : outerEnvironment;

  // Aktualisiere Closures im Heap.
  combineWith(addresses, bindings, (address, bind) -> {
    List<Value> capturedValues = new ArrayList<>();
    for (Variable free : bind.lambda.freeVariables())
      capturedValues.add(rhsEnvironment.get(free));

    heap.update(address,
      new Closure(bind.lambda, capturedValues));
  });
  return localEnvironment;
}
```

Listing 4.5: Funktion `allocateAll` zum Anlegen rekursiver Namensbindungen

4.6 Fallunterscheidungen

Wird im Auswertungszustand der Maschine auf eine Fallunterscheidung als Ausdruck getroffen, wird nicht sofort zur passenden Alternative gesprungen. Die Alternativen werden lediglich als Continuation auf den *Return-Stack* geschoben und Maschine führt mit der Auswertung des untersuchten Ausdrucks fort.

$$(4)\quad \begin{array}{lll} & \mathit{Eval}\ (\texttt{case}\ e\ \texttt{of}\ \mathit{alts})\ \rho\ \mathit{as} & \mathit{rs}\ \mathit{us}\ h\ \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \mathit{as}\ (\mathit{alts}, \rho) : \mathit{rs}\ \mathit{us}\ h\ \sigma \end{array}$$

Abbildung 4.6: Regel zur Auswertung von Fallunterscheidungen

Die Implementierung in Java schiebt ebenso nur eine Continuation auf den Stack und gibt den neuen Maschinenzustand als Ergebnis zurück.

```
public State visit(CaseExpression expression) {
  returnStack.push(
      new Continuation(expression.alternatives, rho));
  return new Eval(expression.scrutinized, rho);
}
```

Die tatsächliche Auswahl einer Alternative geschieht erst nachdem die Auswertung des untersuchten Ausdrucks abgeschlossen ist. Dann einen speziellen Zustand übergegangen, um zur Fallunterscheidung zurückzukehren. Dieser Zustand tritt ein, wenn die Anwendung eines Konstruktors ausgewertet wird, wie die Regel in Abbildung 4.7 zeigt.

$$(5)\quad \begin{array}{lll} & \mathit{Eval}\ (c\ \mathit{xs})\ \rho & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h\ \sigma \\ \Longrightarrow & \mathit{ReturnCon}\ c\ (\mathit{val}\ \rho\ \sigma\ \mathit{xs}) & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h\ \sigma \end{array}$$

Abbildung 4.7: Regel zur Auswertung von Konstruktoranwendungen

Abbildungen 4.8, 4.9 und 4.10 zeigen die involvierten Regeln zur Auswahl einer Alternative. Da diese drei Regeln zusammen wirken, um die korrekte Alternative auszuwählen, werden sie zu einer Methode in Java zusammengefasst. Die Continuation wird zunächst vom Stack entfernt und anschließend die darin gespeicherten Alternativen der Reihe nach untersucht, ob deren Konstruktor dem Konstruktor aus dem Maschinenzustand entspricht. Falls eine passende Alternative gefunden wird, werden die Variablen der Alternative entsprechend Regel 6 gebunden und die in die Auswertung des Ausdrucks der Alternative übergegangen. Auf diese Weise wird die Schleife über die Alternativen vorzeitig abgebrochen.

Besitzt jedoch keine der Alternativen einen passenden Konstruktor, so wird die Standardalternative untersucht. Zunächst wird auf den Fall aus Regel 8 geprüft. Tritt dieser auf, wird das gesamte Ergebnis der Auswertung an einen Namen gebunden statt lediglich einzelne Variablen zu extrahieren. Dafür muss eine Closure angelegt werden, die den Konstruktor und die angewandten Werte speichert. Regel 7 hingegen bindet keinen Namen und setzt die Ausführung wie in Regel 8 mit der Auswertung des Ausdrucks aus der Alternative fort.

```
public record ReturnConstructor(Constructor constructor,
                List<Value> arguments) implements State {
  public State transfer(Machine machine) {
    final var continuation = machine.returnStack.pop();

    // Suche nach passender Alternative.
    for (Alternative alt : continuation.alternatives())
      if (areEqual(this.constructor, alt.constructor)) {
        // Passende Alternative gefunden.
        combineWith(alt.arguments, this.arguments,
          continuation.savedEnvironment()::put);
        return new Eval(alt.expression,
          continuation.savedEnvironment());
      }

    // Verwende Standard-Alternative.
    if (continuation.defaultAlternative()
         instanceof DefaultBindingAlternative def) {
      Address address = machine.heap
        .allocate(standardConstructorClosure());
      continuation.savedEnvironment()
        .put(def.variable, address);
      return new Eval(def.expression,
        continuation.savedEnvironment());

    } else if (continuation.defaultAlternative()
         instanceof DefaultFallthroughAlternative def) {
      return new Eval(def.expression,
        continuation.savedEnvironment());
    }
  }
}
```

Listing 4.6: Implementierung der Auswahl einer Alternative

$$(6)\quad \boxed{\begin{array}{ll} & \mathit{ReturnCon}\ c\ ws \quad as\ (\ldots;\ c\ vs\ \texttt{->}\ e;\ \ldots,\ \rho) : rs\ us\ h\ \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho[vs \mapsto ws]\ as \qquad\qquad\qquad\qquad\qquad\quad rs\ us\ h\ \sigma \end{array}}$$

Abbildung 4.8: Regel zur Auswahl einer Alternative

$$(7)\quad \boxed{\begin{array}{ll} & \mathit{ReturnCon}\ c\ ws\ as \left(\begin{array}{l} c_1\ vs_1\ \texttt{->}\ e_1; \\ \ldots; \\ c_n\ vs_n\ \texttt{->}\ e_n; \\ \texttt{default ->}\ e_d \end{array}, \rho\right) : rs\ us\ h\ \sigma \\ \text{such that} & c \neq c_i \qquad (1 \leq i \leq n) \\ \Longrightarrow & \mathit{Eval}\ e_d\ \rho \qquad as \qquad\qquad\qquad\qquad rs\ us\ h\ \sigma \end{array}}$$

Abbildung 4.9: Regel zur Auswahl der Standardalternative

$$(8)\quad \boxed{\begin{array}{ll} & \mathit{ReturnCon}\ c\ ws\ as \left(\begin{array}{l} c_1\ vs_1\ \texttt{->}\ e_1; \\ \ldots; \\ c_n\ vs_n\ \texttt{->}\ e_n; \\ v\ \texttt{->}\ e_d \end{array}, \rho\right) : rs\ us\ h\ \sigma \\ \text{such that} & c \neq c_i \qquad (1 \leq i \leq n) \\ \Longrightarrow & \mathit{Eval}\ e_d\ \rho[v \mapsto a]\ as \qquad\qquad\qquad\qquad rs\ us\ h'\ \sigma \\ \text{where} & h' = h[a \mapsto (vs\ \texttt{\textbackslash n \{\}->}\ c\ vs)\ ws] \\ & vs \text{ is a sequence of arbitrary distinct variables} \\ & \mathit{length}(vs) = \mathit{length}(ws) \end{array}}$$

Abbildung 4.10: Regel zur Auswahl der Standardalternative mit Namensbindung

Für primitive Zahlenwerte wird ein ähnlicher Zustand `ReturnInt` bereitgestellt, welcher sich analog zu `ReturnConstructor` verhält. Statt die verschiedenen Alternativen auf passende Konstruktoren zu überprüfen, wird nach einem passenden Zahlenwert gesucht (Regel 11). Die Regeln zur Auswahl von Standardalternativen (Regel 12 und Regel 13) sind äquivalent zu denen im Zustand `ReturnConstructor`. Die einzige Besonderheit bei der Auswertung von primitiven Zahlenwerten ist, dass die Auswertung einer Variable, die einen primitiven Zahlenwert speichert, in den Zustand `ReturnInt` übergeht. Dies erklärt die Einschränkung in Abbildung 4.3 aus Abschnitt 4.4.

4.7 Aktualisierbare Closures

Zum Aktualisieren von Closures auf dem Heap müssen mehrere Regeln zusammenspielen. Einerseits wird beim Betreten einer aktualisierbaren Closure ein entsprechender Rahmen auf den Update-Stack geschoben. Andererseits muss der Rahmen vom Stack wieder entfernt werden, wenn ein Update ausgelöst wird.

$$
(15)\quad
\begin{array}{lll}
 & \mathit{Enter}\ a\ \ \mathit{as}\ \ \mathit{rs} & \mathit{us}\ \ h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash u}\ \texttt{\{\}}\ \texttt{->}\ e)\ \ \mathit{ws}_f]\ \ \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \ \rho\ \ \{\}\ \ \{\}\ \ (\mathit{as}, \mathit{rs}, a) : \mathit{us}\ \ h & \sigma \\
\text{where } \rho = [\mathit{vs} \mapsto \mathit{ws}_f] & &
\end{array}
$$

Abbildung 4.11: Regel zum Betreten einer aktualisierbaren Closure

Regel 15 in Abbildung 4.11 stellt das Betreten einer aktualisierbaren Closure dar. Diese Regel ist ähnlich zur Regel zum Betreten einer nicht-aktualisierbaren Closure mit dem Unterschied, dass ein Update-Rahmen auf den entsprechenden Stack gelegt wird. Das Erstellen dieses Rahmens leert den Argument- und Return-Stack. Die Inhalte beider Stacks werden im Rahmen gespeichert zusammen mit der Adresse der aktualisierbaren Closure. Dieses Verhalten lässt sich durch Anpassen der existierenden Implementierung mit minimalen Aufwand erreichen, indem eine einzelne Verzweigung ergänzt wird:

```
if (closure.code().isUpdateable) {
  machine.updateStack.push(new UpdateFrame(
    machine.argumentStack, machine.returnStack, address));
  machine.argumentStack = emptyStack();
  machine.returnStack = emptyStack();
}
 ...
```

Der schwierigere Teil ist das Anwenden einer Aktualisierung, wenn die Auswertung einer aktualisierbaren Closure abgeschlossen ist. Es existieren zwei Fälle, in denen der Abschluss einer Auswertung erkannt und somit deine Aktualisierung ausgelöst wird.

1. Wird der Wert einer Closure als Konstruktor bekannt, so folgt anschließend der Versuch, eine Continuation vom Return-Stack zu holen, um die Auswahl einer Alternative auszuführen. Wenn der Return-Stack leer ist, kann dieser Versuch nur scheitern. Diese Situation wird dann als Aktualisierung erkannt.

2. Ist der Wert einer Closure eine Funktion, versucht diese die Funktionsargumente vom Argument-Stack zu konsumieren. Sind keine Argumente auf dem Stack vorhanden, so scheitert dieser Versuch ebenso, was als Aktualisierung erkannt wird.

Diese beiden Ereignisse werden als Regel 16 und Regel 17 dargestellt.

(16)

$$\begin{array}{lll} & \mathit{ReturnCon}\ c\ ws\ \ \{\}\ \ \{\}\ (as_u, rs_u, a_u) : us\ \ h & \sigma \\ \Longrightarrow & \mathit{ReturnCon}\ c\ ws\ \ as_u\ \ rs_u & us\ \ h_u\ \ \sigma \\ \text{where} & vs \text{ is a sequence of arbitrary distinct variables} & \\ & \mathit{length}(vs) = \mathit{length}(ws) & \\ & h_u = h[a_u \mapsto (vs\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ c\ vs)\ ws] & \end{array}$$

Abbildung 4.12: Regel zur Erkennung von Aktualisierungen bei Konstruktoren

(17)

$$\begin{array}{lll} & \mathit{Enter}\ a\ \ as\ \ \{\}\ (as_u, rs_u, a_u) : us\ \ h & \sigma \\ \text{such that} & h\ a = (vs\ \texttt{\textbackslash n}\ xs\ \texttt{->}\ e)\ ws_f & \\ & \mathit{length}(as) < \mathit{length}(xs) & \\ \Longrightarrow & \mathit{Enter}\ a\ \ as +\!\!+ as_u\ \ rs_u & us\ \ h_u\ \ \sigma \\ \text{where} & xs_1 +\!\!+ xs_2 = xs & \\ & \mathit{length}(xs_1) = \mathit{length}(as) & \\ & h_u = h[a_u \mapsto ((vs +\!\!+ xs_1)\ \texttt{\textbackslash n}\ xs_2\ \texttt{->}\ e)\ (ws_f +\!\!+ as)] & \end{array}$$

Abbildung 4.13: Regel zur Erkennung von Aktualisierungen bei Funktionen

Wie bereits zu erkennen ist, sind diese Regeln vergleichsweise komplex und benötigen einige Hilfsdefinitionen, die größtenteils dazu dienen, die Stacks korrekt wiederherzustellen und die korrekte Anzahl an Werten für den Folgezustand zur Verfügung zu stellen. Um beide Fälle in der Implementierung zu erkennen, wird jeweils eine Verzweigung ergänzt.

Listing 4.7 stellt dar, wie im Zustand `ReturnConstructor` überprüft wird, ob der Return-Stack ein Element zum Entfernen enthält (siehe Regel 16). Ist dies

nicht der Fall, wird der Konstruktor samt Parameter bei der Ersetzung eingesetzt. Der Code zum Einrichten der Closure ist hierbei identisch zu dem, der ausgeführt wird, wenn die Standardalternative im Zustand `ReturnConstructor` einen Namen bindet.

```
if (machine.returnStack.isEmpty()) {
  final UpdateFrame frame = machine.updateStack.pop();

  machine.argumentStack = frame.argumentStack();
  machine.returnStack = frame.returnStack();
  machine.heap.update(frame.address(),
   standardConstructorClosure());

  return this;
}
```

Listing 4.7: Erkennung von Aktualisierungen bei Konstruktoren

Bei der Übersetzung von Regel 17, wie sie in Listing 4.8 zu sehen ist, wird ein großteil der Codezeilen für die Manipulation der verschiedenen Listen verwendet. Was sich als Muster kurz und präzise ausdrücken lässt, lässt sich in Java lediglich über das Zusammenspiel von mehreren Listenprimitiven implementieren. Da diese Codezeilen nur schwer nachvollziehbar sind und Abbildung 4.13 bereits deren Bedeutung darstellt, werden sie hier weggelassen. Die Closure, die zur Ersetzung erstellt wird, kopiert dabei den Rumpf aus der Continuation. Lediglich Parameter, freie Variablen und deren Werte werden angepasst.

```
if (machine.argumentStack.size() < closure.code().parameters
   .size()) {
  final List<Variable> parameters = ...;
  final List<Variable> freeVariables = ...;
  final List<Value> boundValues = ...;

  final UpdateFrame frame = machine.updateStack.pop();
  machine.returnStack = frame.returnStack();
  machine.argumentStack.addAll(frame.argumentStack());

  machine.heap.update(frame.address(), new Closure(...));
  return this;
}
```

Listing 4.8: Implementierung der Erkennung von Aktualisierungen bei Funktionen

Als Besonderheit bei diesen beiden Regeln ist festzustellen, dass sich die Code-Komponente nicht ändert. Bei der Implementierung in Java wird dies besonders deutlich, da `return this` den nächsten Codezustand und somit den aktuellen Zustand unverändert bereitstellt. Grund dafür ist, dass der Wert, der durch die Aktualisierung einer Closure entsteht, das Ergebnis einer anderen aktualisierbaren Closure sein könnte. Daher wird erneut auf die verschiedenen Fälle geprüft, um eventuell eine weitere Closure zu aktualisieren.

4.8 Ausführung

Da die Code-Komponente der Maschine *tagless* mit Hilfe des Polymorphismus in Java implementiert wurde, ist die tatsächliche Ausführung der Maschine einfach umzusetzen. Ein Ausführungsschritt aktualisiert die Komponente des Zustands durch den Ergebniswert der `State transfer(Machine)` Methode, die auf dem aktuellen Zustand aufgerufen wird.

```
public void step() {
  // Auswahl der Aktion erfolgt automatisch durch Sprung zur
    Implementierung.
  state = state.transfer(this);
}
```

Das Ende der Ausführung wird mit dem Abschluss der Auswertung des Startzustands erreicht. Den Regeln entsprechend wird hier eine Funktion ohne hinreichende Anzahl an Argumenten aufgerufen oder einer der beiden Zustände `ReturnInt` oder `ReturnConstructor` wird aktiv, obwohl der Continuation-Stack leer ist. Beides sind Indizien, dass eine Aktualisierung aussteht, jedoch existiert dann keine Closure mehr, die aktualisiert werden könnte. Dieser Fall kann abgefangen werden, um das Ende der Ausführung festzustellen.

5 Fazit

Mit den Regeln aus Kapitel 4 und den vorgestellten Sprachkonstrukten aus Kapitel 3 ist es nun möglich, in der STG-Sprache geschriebene, nicht-strikte Programme auszuwerten. Die Ausführung erfolgt dabei entsprechend der Spezifikation, ermöglicht die Integration von primitiven Java-Operationen und arbeitet korrekt auf allen unterstützten Werten von primitiven Zahlen, über einfache Datentypen bis hin zu Funktionen und verzögert ausgewerteten oder sogar endlosen Datenstrukturen.

Die vollständige Implementierung in Java ist auf GitHub verfügbar.[1] Im selben Projekt befinden sich neben den verschiedenen Quelldateien auch eine Prelude, die vor der Ausführung von STG-Programmen geladen werden kann, um einige häufig verwendete Funktionen bereitzustellen.

Das Projekt erfüllt dabei die Ansprüche und Ziele, die zu Beginn gesetzt wurden und präsentiert eine einfache aber funktionstüchtige Implementierung einer virtuellen STG-Maschine. Zusätzlich werden durch die Prelude die Eigenschaften der STG-Sprache als Kernsprache hervorgehoben und exemplarisch gezeigt, wie höhere Abstraktionsebenen in die maschinennahe Sprache eingeführt werden können. Das Hauptziel, die Semantik der STG-Maschine korrekt abzubilden wurde erreicht. Darüber hinaus werden einige der Erweiterungen, die von der STG vorgestellt werden, implementiert. Neben der *spineless* Darstellung eines Programms, die durch Verwendung der STG „geschenkt“ ist, wird auch die *tagless* Darstellung implementiert. Die verschiedenen Maschinenzustände werden nicht durch explizite Tags unterschieden. Stattdessen werden sie durch den Aufruf einer Methode dynamisch zur Laufzeit ausgewählt. Zudem werden auch Optimierungen, wie etwa das Aktualisieren von ausgewerteten Closures, unterstützt.

[1] `https://github.com/Niklas-Deworetzki/java-stg`

5.1 Wichtige Erweiterungen

Obwohl die Kernfunktionalität der STG implementiert ist und auch die besonderen Eigenschaften der Maschine unterstützt werden, existieren einige Erweiterungen und diskutierte Optimierungen aus [14], die nicht implementiert sind.

Viele der möglichen Erweiterungen, die sich in die STG-Maschine einbauen lassen, sind in erster Linie dazu gedacht, die Programmierung zu erleichtern oder den produktiven Betrieb zu ermöglichen. So kann beispielsweise ein Foreign Function Interface bereitgestellt werden, um mit existierenden Bibliotheken anderer Programmiersprachen zu interagieren [15]. Die Implementierung dieser Schnittstelle ist dabei ähnlich wie die Integration eingebauter, primitiver Funktionen.

Eine weitere hilfreiche Erweiterung baut auf das Aktualisieren von Closures auf. Wird eine Closure nachdem sie betreten wurde, aber bevor sie vollständig ausgewertet wird, durch einen speziellen Wert ersetzt, lassen sich Fehler zur Laufzeit entdecken. Ein sogenanntes *Black Hole* lässt sich für eine betretene Closure einsetzen, mit der Eigenschaft, dass das Betreten eines solchen Loches einen Fehler wirft. Der Fehlerfall tritt nur dann auf, wenn während der Auswertung einer Closure, die selbe Closure erneut betreten wird. Das bedeutet, dass ein Wert von sich selbst abhängig ist und nicht berechnet werden kann [14].

Anstelle eines Black Holes, können aber auch andere besondere Darstellungen für Closures eingesetzt werden. Beispielsweise ist es möglich, betretene Closures durch Synchronisierungsblöcke zu ersetzen. Dadurch können mehrere Closures gleichzeitig betreten und in unterschiedlichen Threads ausgewertet werden. Ist die Auswertung abgeschlossen sorgen die Synchronisierungsblöcke dafür, dass die verschiedenen Threads wieder zusammengeführt werden. Auf diese Weise kann unsichtbar für einen Nutzer Parallelisierung implementiert werden [14].

Die Wohl wichtigste Erweiterung, die zunächst notwendig ist, um die produktive Verwendung der Implementierung überhaupt in Betracht ziehen zu können, ist die Implementierung eines Garbage Collectors. Die Sprachsemantik definiert nur ein Konstrukt für die Speicherallokation. Eine kontrollierte Speicherfreigabe ist nicht vorgesehen. Stattdessen werden einige Algorithmen präsentiert, die automatisch ungenutzte Closures auf dem Heap erkennen und freigeben können [14]. Wird ein solcher Algorithmus angestoßen, wenn nicht mehr genügend Speicher verfügbar ist oder läuft er dauerhaft im Hintergrund, ist es möglich, nicht benötigten Speicher freizugeben und auf dem Heap lediglich Closures zu halten, die für die Auswertung relevant sind.

5.2 Schwachstellen der Implementierung

Auch wenn sie entsprechend der Beschreibung arbeitet, existieren einige Schwachstellen in der vorgestellten Implementierung. Diese Beziehen sich hauptsächlich auf die Ausführung und die Nutzerfreundlichkeit der Maschine.

Ein Beispiel hierfür ist der Abschluss der Ausführung. Die Maschine selbst sieht keinen Endzustand vor, in dem die Ausführung abgeschlossen ist und der von außerhalb eindeutig als solcher Erkannt werden kann. Stattdessen wird – wie in Abschnitt 4.8 dargestellt – erkannt, wenn beim Ersetzen einer Closure auf den leeren Update-Stack zugegriffen wird. Liegt in einem solchen Fall ein *Return Integer* oder *Return Constructor*-Zustand vor, wird der von diesen zurückgegebene Wert als Ergebnis der Maschine angezeigt. Dabei wird lediglich die Zahl als Text formatiert oder der Name des Konstruktors ausgegeben. Komponenten von Datenstrukturen liegen lediglich als Adresse auf dem Heap und womöglich unausgewertet dem Konstruktor vor. Eine komplexere Ausgabe ist also nicht ohne Weiteres möglich.

Auch vor dem Erreichen eines Programmendes können nutzerunfreundliche Eigenschaften der Implementierung beobachtet werden. Liegt in einem ausgeführten STG-Programm ein Fehler vor, so ist das Verhalten der Maschine oftmals unerwartet (wenn auch Korrekt bezüglich der Maschinensemantik). Fehler werden an scheinbar willkürlichen Stellen bekannt und durch die verzögerte Auswertung teilweise auch erst in späteren Programmabschnitten relevant. Ohne einen Debugger und eine intuitive Darstellung von komplexen Werten wird das Debugging oft zu einem Ratespiel.

Der Hauptnutzen bei der Implementierung liegt wohl im akademischen Wert und in der gesammelten Erfahrung. Doch auch hier werden einige Kapitel aus [14] ausgelassen. In einer Sprache, die mehr Kontrolle über Speicherlayout und Speicherallokation bietet, wäre es möglich, Diskussionen über die Darstellung und das Speicherlayout von Closures sowie die Speicherverwaltung mit einem Garbage Collector nachzuvollziehen. Eine vollständige Implementierung all dieser Aspekte sprengt jedoch den Rahmen (und auch den Fokus) dieser Veranstaltung bei Weitem.

5.3 Vergleich mit anderen Implementierungen

Letztlich soll die entstandene Implementierung im Vergleich zu bereits existierendem Material zur STG-Maschine eingeordnet werden. Viele weitere Implementierungen der STG-Maschine sind nicht bekannt. Jedoch wird sie gerade im Kontext von Haskell benutzt und dementsprechend auch Weiterentwickelt. Hier ist als größte Erweiterung ein anderer Ansatz bei der Übergabe von Funktionsargumenten zu

nennen, wodurch gerade bei Funktionen höherer Ordnung für die häufigsten Anwendungsfälle die Performance verbessert wird [13].

Als andere Implementierung der STG-Maschine existiert das Projekt *STGi* [11]. Geschrieben in Haskell steht dieser Interpreter für die STG-Sprache als Lehrbeispiel zur Verfügung. Mit einer textuellen Darstellung von Werten, Zuständen und Zustandsübergängen sowie implementierten Erweiterungen wie Black Holes, Garbage Collection und der Semantik aus [13], liegt der Fokus dieser Implementierung jedoch darauf, die Ausführung interaktiv zu begleiten. Die Arbeit, welche die STG bei der Übersetzung in imperative Anweisungen betreibt, wird bei der Implementierung in Haskell nicht deutlich.

Die wohl wichtigste STG-Implementierung befindet sich ebenfalls in der Domäne von Haskell. Der sogenannte GHC Haskell Compiler verwendet intern die STG, um Maschinencode entweder nativ oder mittels LLVM zu erzeugen. Dabei bietet diese Implementierung nicht nur Unterstützung für viele Erweiterungen, Optimierungen, Tooling und Profiling, sondern stellt auch eine produktionsreife Ausführungsumgebung für Haskell bereit. Als de facto Standard für Haskell, ist hier die wohl ausgereifteste Implementierung zu finden. Geschrieben in Haskell, über Jahre gewachsen und mit Unterstützung für Optimierungen ist sie jedoch nicht als Lehrbeispiel geeignet.

Zusammenfassend ist dieses Projekt eher als Lern- und Spielzeugprojekt zu einzuordnen. Der Hauptzweck der Implementierung ist es, innerhalb des Kurses *CS5341 – Kernel-Architekturen in Programmiersprachen* Erfahrungen zu sammeln, neue Berechnungsmodelle zu untersuchen und den eigenen Horizont zu erweitern.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] Lennart Augustsson und Thomas Johnsson. „Parallel Graph Reduction with the (v , G)-Machine“. In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, 1989, S. 202–213.

[2] Max Bolingbroke. „Generated Code – What is STG, Exactly?“ In: *The GHC Wiki* (Apr. 2010). URL: `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/generated-code` (besucht am 04. 01. 2022).

[3] E. Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] Adele Goldberg. *Smalltalk-80 – The Interactive Programming Environment*. Reading, Mass: Addison-Wesley, 1983.

[5] Paul Graham. *On Lisp – Advanced Techniques for Common Lisp*. Prentice Hall, 1993.

[6] Paul Hudak u. a. „A History of Haskell – Being Lazy with Class“. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. 12. San Diego, California, 2007, S. 1–55. DOI: 10.1145/1238844.1238856.

[7] Susann Ragsdale James S. Miller. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, 2003.

[8] Richard B. Kieburtz. „The G-machine – A Fast, Graph-reduction Evaluator“. In: *Functional Programming Languages and Computer Architecture*. 1985, S. 400–413.

[9] Hugh Kingdon, David R. Lester und Geoffrey L. Burn. „The HDG-Machine – A Highly Distributed Graph-Reducer for a Transputer Network“. In: *The Computer Journal* 34 (1991), S. 290–301.

[10] Tim Lindholm u. a. *The Java Virtual Machine Specification – Java SE 17 Edition*. Spezifikation. Oracle America, Inc., 2021.

[11] David Luposchainsky u. a. *STGi – STG Interpreter*. 2021. URL: `https://github.com/quchen/stgi/tree/2920b7c1f83f5516030d1806ce4913d46a` (besucht am 19. 01. 2022).

[12] Martin Odersky und Tiark Rompf. „Unifying Functional and Object-oriented Programming with Scala“. In: *Communications of the ACM* 57.4 (2014), S. 76–86. DOI: 10.1145/2591013.

[13] Simon Peyton Jones. „How to Make a Fast Curry – Push/Enter vs Eval/Apply“. In: *International Conference on Functional Programming*. 2004, S. 4–15.

[14] Simon Peyton Jones. „Implementing Lazy Functional Languages on Stock Hardware – The Spineless Tagless G-machine“. In: *Journal of Functional Programming* 2 (1992), S. 127–202.

[15] Simon Peyton Jones. „Tackling the Awkward Squad – Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell“. In: *Engineering theories of software construction*. 2001, S. 47–96. ISBN: 1-58603-1724.

[16] Wikibooks, The Free Textbook Project. *Haskell / Laziness*. 2020. URL: https://en.wikibooks.org/w/index.php?title=Haskell/Laziness&oldid=3676028 (besucht am 05. 01. 2022).

[17] Cees Slot und Peter van Emde Boas. „The Problem of Space Invariance for Sequential Machines“. In: *Information and Computation* 77.2 (1988), S. 93–122. DOI: 10.1016/0890-5401(88)90052-1.

[18] Guy L. Steele. *Common Lisp – The Language*. Digital Press, 1990.

[19] *The LLVM Compiler Infrastructure*. 2022. URL: https://llvm.org/ (besucht am 19. 01. 2022).

[20] Philip Wadler. „Efficient Compilation of Pattern Matching“. In: *The Implementation of Functional Programming Languages* (1987), S. 78–103.

[21] Christopher P. Wadsworth. „The Relation between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus“. In: *SIAM Journal on Computing* 3.5 (1976), S. 488–521.