CS5341 – Kernel-Architekturen in Programmiersprachen

Thema:
The Spineless Tagless G-Machine

Vorgelegt von:	Niklas Deworetzki
Matrikelnummer	5185551
Eingereicht bei Hochschulbetreuer/-in:	Prof. Dr.-Ing. Dominikus Herzberg
Eingereicht am:	17. Januar 2022

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 bei der Implementierung 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. 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. Bei den objektorientierten Programmiersprachen ist Smalltalk als wichtiger Vertreter zu nennen. 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 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. 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, welche dann auf Argumente angewandt, an andere Funktionen übergeben oder an Namen gebunden werden können. 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 ???[citation needed] als wesentlicher Bestandteil in der Implementierung und Übersetzung von Haskell verwendet. Im Ergebnis ist Performance von übersetzten Haskell Programmen häufig vergleichbar mit C.[citation needed]

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 Semantik

in Maschinencode für eine reale Maschine übersetzt wird. 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. Eine ähnliche Architektur bietet LLVM mit einer abstrakten Registermaschine. 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 [2] Einzug in die theoretische Welt der Berechenbarkeit gehalten haben. Die Grundannahmen sind hier, dass Rechenschritte einzelne gleichwertige Instruktionen darstellen, die nacheinander ausgeführt werden und jeweils eine Menge an Registern oder Speicherzellen beeinflussen können.

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 ausgewertete Ausdruck schließlich eine sogenannte Normalform erreicht – eine einheitliche Form des Ausdrucks, die dessen ausgewerteten Wert repräsentiert.

Der Teil *G-Maschine* 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[citation needed]. 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.

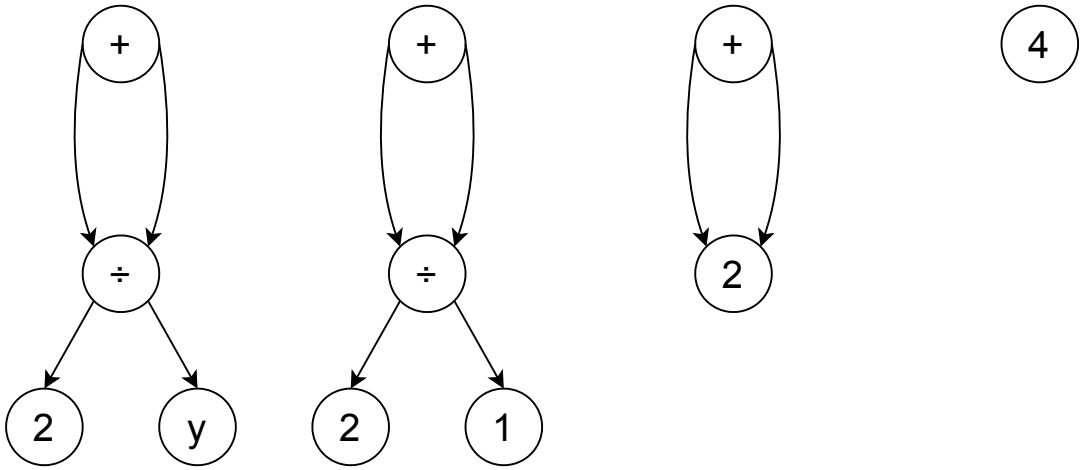

Abbildung 2.1: Schrittweise Redutkion des Ausdrucks $\texttt{let}\ x\ =\ \frac{2}{y}\ \texttt{in}\ x + x$

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 für anonyme Funktionen 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 hier das Definieren einer Normalform, welche die ausgewertete Form eines Ausdrucks darstellt. Die Anwendung von Reduktionsregeln auf einen Teilgraphen in Normalform verändert diesen nicht mehr; ein Ende der Auswertung wurde erreicht. Im Rahmen der STG wird die Weak Head Normal Form (WHNF) verwendet [1]. Die WHNF beschreibt, dass lediglich der Kopf eines Graphen in Normalform vorliegen muss, um ihn als ausgewertet zu betrachten. Dies ist der Fall, wenn der oberste Knoten des Graphen einen Konstruktor 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 oberste Listenelement aussieht, ist dabei für die WHNF irrelevant.

Problematisch wird trotz der WHNF dennoch die Auswertung von unendlichen Graphen oder Graphen, die einen Zyklus enthalten. Gerade endlose Graphen erweisen sich als Herausforderung, wenn man den naiven Ansatz verfolgt und den gesamten Programmgraphen als Datenstruktur im Speicher hat. Diesen Problem-

fall behandelt das *S* in 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 der Teil, der für die aktuelle Auswertung relevant ist.

Die letzte Erweiterung der STG, wird durch das *T* 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 einen Knoten markieren und dessen Ausprägung bestimmen [citation needed]. 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 einheitliche Darstellung 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.

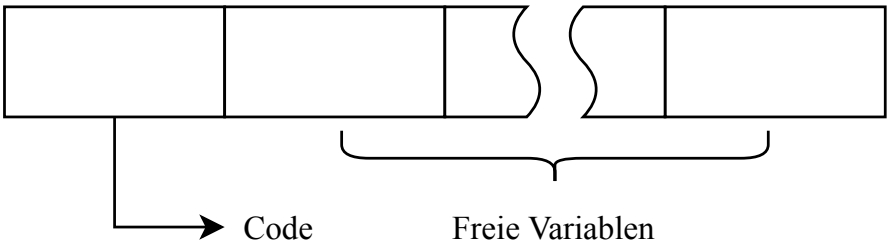

Abbildung 2.2: Aufbau einer Closure

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, wessen Zusammenspiel einen Graphen bilden. 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 Lambda Formen

Lambda Formen entsprechen den klassischen Lambda Ausdrücken, 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.

Zunächst existiert eine weitere Liste an Variablen bei der Definition einer Lambda Form. Diese Liste 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. Da die Variablen nicht in dem Ausdruck definiert werden, aber durch eine umschließende Definition sichtbar sind, muss ein Verweis auf den entsprechenden Wert gespeichert werden. Da dies Speicherplatz benötigt und somit die Ausführung der

STG-Maschine beeinflusst, werden die freien Variablen explizit angegeben.

Weiterhin fällt auf, dass eine *Update Flag* als Teil der Lambda Form 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 Lambda Form eine Funktion beschreibt, oder der Ausdruck im Rumpf bereits in der WHNF ist [citation needed].

3.2 Let Bindungen

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

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

Hinsichtlich der Maschinensemantik beschreiben Bindungsausdrücke immer Speicherallokation. Für jede gebundene Lambda Form 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 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 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[citation needed].

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 auf diesen – wie alle weiteren Operationen auch – als Closures 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 Zahlenwerte aus

deren Closures, die eigentliche Auswertung der Addition, das Anlegen einer neuen Closure für den Ergebniswert und das anschließende Ablegen des Ergebnisses.

Die STG bietet über primitive Werte eine Möglichkeit, verfügbare Maschinendatentypen und Maschineninstruktionen einer echten Hardware direkt in die 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#
```

Beschreibt in der Haskell-Syntax die Definition eines 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. In diesem Programmteil wird die Konvention angewandt, Bezeichner für primitive Variablen mit dem Suffix `#` zu versehen, wie es bei den primitiven Arithmetischen Operationen 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 [citation needed] 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 eine imperative Programmiersprache wie 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 jedoch während der gesamten Laufzeit nicht verändert.

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-Frame) 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 Record erstellt, welcher einer gemeinsame Schnittstelle implementiert.

Listing 4.1: Darstellung von Werten in Java

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

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

Ähnlich wird bei den Ausprägungen des Maschinencodes vorgegangen. Hier existieren insgesamt vier Ausprägungen: **Eval**, **Enter**, **Return Constructor** und **Return Integer**.

Listing 4.2: Darstellung der Ausprägungen des Maschinencodes in Java

```
public sealed interface Code {
  record Eval(Expression expression,
    Map<Variable, Value> locals) implements Code {}

  record Enter(int address) implements Code {}

  record ReturnConstructor(Constructor constructor,
    List<Value> arguments) implements Code {}

  record ReturnInteger(int integer) implements Code {}
}
```

4.2 Voraussetzungen zur Auswertung

Funktion val

4.3 Funktionsanwendungen

4.4 Namensbindungen

4.5 Fallunterscheidungen

5 Ergebnisse

Abbildungsverzeichnis

Literaturverzeichnis

[1] 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).

[2] 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: https://doi.org/10.1016/0890-5401(88)90052-1.