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

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:	5. 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 (CIL), 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.

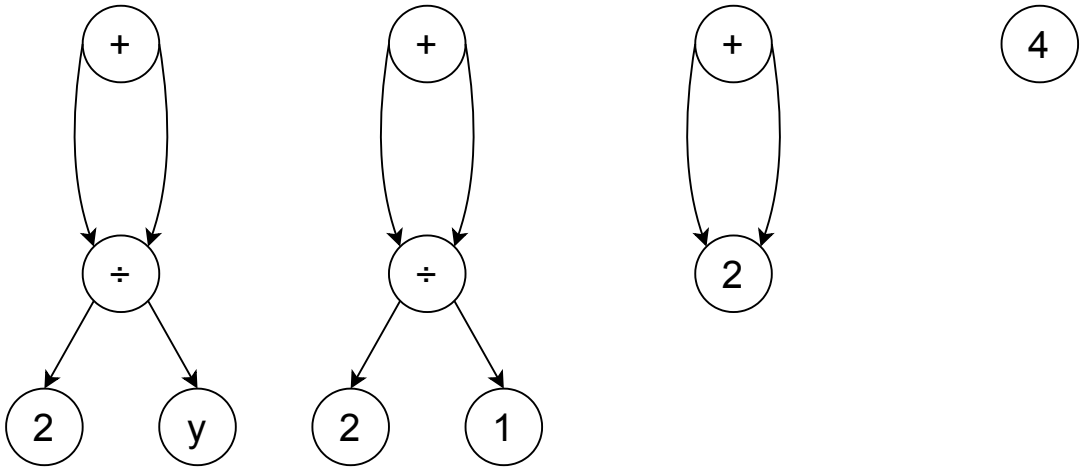

Abbildung 2.1: Schrittweise Redutkion des Ausdrucks `let` $x = \frac{2}{y}$ `in` $x + x$

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 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.

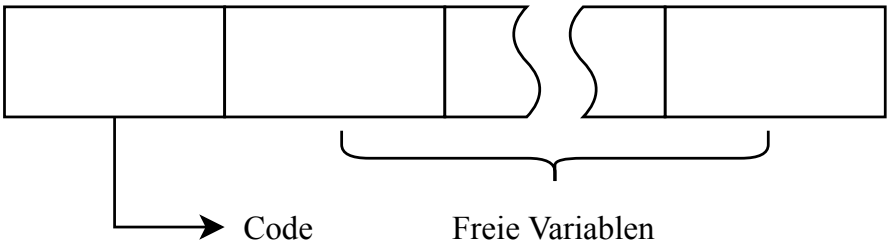

Abbildung 2.2: Aufbau einer Closure

3 Die STG-Sprache

Sprache gibt Beschreibung der STG-Maschine an. Hier ist auch Unterschied zu anderen Maschinensprachen deutlich. Keine Instruktionen, stattdessen wird Programmgraph als funktionales Program beschrieben.

Funktionale Programmiersprache dennoch mit Einschränkungen verbunden, die Ausdrucksstärke reduzieren. Ziel ist nicht angenehmes Programmieren sondern Nähe zur Semantik.

Verschiedene Sprachkonzepte werden vorgestellt und informell mit Bedeutung in der Maschine verknüpft.

Überblick mit Grammatik. Aussehen ähnelt Haskell, wird auch so dargestellt. Verwendung von Einrückung und Zeilenumbrüchen zur Abgrenzung. Interessante Anmerkung: Keine Typisierung im Vergleich zu Haskell.

3.1 Lambda

Viele Besonderheiten: Freie Variablen (benötigt zur Berechnung von Speicherbedarf der Closure) Update-Flag

Ansonsten wir gewohnt: Parameter Rumpf

3.2 Let-Bindings

Let und Letrec. Erstellt immer Lambdas, direkter Kontakt zu lazy.

Bedeutung ist Speichern auf dem Heap (Heap-Allokation)

3.3 Anwendungen

Funktionen, Konstruktoren, Primitive Operationen. Eta-Expansion nicht wie in Haskell.

Argumente immer Atome: Variablen oder Primitive Zahlen. Erfordert anlegen auf Heap via Let für komplexe Parameter. Einschränkung der Ausdrucksstärke (tatsächlich Ausdruck eingeschränkt), dafür jedoch beschränkte Komplexität in Maschine. Bei Aufruf müssen Parameter nicht untersucht werden: Alle direkt in atomarer Form.

3.4 Fallunterscheidungen

Auswertung nur hier. Zwingend nötig, da WHNF den Konstruktor "kennt", um Alternative zu bestimmen. Somit wichtiges Konstrukt.

Unterscheidung in Algebraische Alternativen und Primitive Alternativen.

Immer vorhanden: Defaults. Passt niemand wird ein Fehler geworfen.

3.5 Primitive

Warum sind Primitive direkt in der Sprachbeschreibung enthalten? Macht Konzept rund, keine Mehraufwand für Maschinenworte. Erhöht Performance, da keine Lazyness und direkt Maschinenworte.

Einpacken in Lazyness nur mit minimaler Abstraktionsebene.

4 Implementierung

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.