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:	25. Dezember 2021

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, sogenannte *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 Markomechanismen 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 werden. 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 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

STG beschreibt sowohl eine abstrakte Maschine als auch eine kleine Programmiersprache, die zur Programmierung ebendieser Maschine verwendet 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, welche die effiziente praktische Umsetzung betreffen. Zur Laufzeit muss zwischen ausstehenden oder unterbrochenen Auswertungen 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. 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 zu entsprechen 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

Wie funktioniert STG? Vergleich zu anderen Ausführungsmodellen von abstrakten Maschinen, die Instruktionen nacheinander Ausführen (RAM, Stackmaschine JVM, Turing-Maschine) ist Modell näher an Lambda-Kalkül, wo Reduktionsschrit-

te durchgeführt werden. Das Programm als Graph wird schrittweise reduziert, bis Ergebnis vorliegt.

Darstellung unterscheidet nicht mehr zwischen Funktionen und Werten, alles sind Closures. Unterscheidung zwischen Wert und Thunk [citation needed] Endlose Graphen dennoch problematisch. Hierfür wird zu Normalform ausgewertet (standardtechnik). WHNF zählt als ausgewerteter Ausdruck.

Gemeinsam kann sowohl Lazy-Evaluation als auch Funktionen höherer Ordnung problemlos ausgewertet werden. Genaue Umsetzung des Modells in Kapitel 2[citation needed].

2.2 Erweiterung STG

G-Machine steht für Maschine basierend auf Graphenreduktion. Ebenfalls wird in WHNF ausgewertet. Welche Änderungen macht S und T?

Es handelt sich um Optimierungen. Erste Graphenmaschinen waren nicht sehr performant, zielten auf besondere Hardware ab (LISP-Maschinen, G-Maschine). Graph als komplexe Datenstruktur muss dargestellt werden. Naive Darstellung ist mit Overhead und Speicherverbrauch verbunden. Ebenfalls muss unterscheidung zwischen Wert und Thunk getroffen werden. Hierfür wurden Tags herangezogen, die Closures markieren.

STG lässt beides weg, um effizienter zu sein. Spineless beschreibt ohne Graphen (Spine). Stattdessen wird über Sprünge im Code oder Zeiger auf dem Heap verwiesen. Tagless bedeutet keine Tags, stattdessen einheitliche Darstellung der Closure, angelehnt an OOP/VTables.

Darum auch keine Unterscheidung zwischen Wert und Thunk. Alles in Closures. Auswertung einer Closure wird immer als Entern der Closure bezeichnet, Sprung zu Code dieser. Code wertet aus oder gibt Ergebniswert zurück.

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