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:	23. Dezember 2021

Inhaltsverzeichnis

Inhaltsverzeichnis **II**

1 Einleitung **1**
1.1 Graphenreduktion als Ausführungsmodell 1
1.2 Erweiterung STG 2

2 STG-Sprache **3**
2.1 Lambda 3
2.2 Let-Bindings 3
2.3 Anwendungen 4
2.4 Fallunterscheidungen 4

3 Implementierung **5**

4 Ergebnisse **III**

Abbildungsverzeichnis **IV**

1 Einleitung

Funktionale Programmiersprachen: Deklarativ & Funktionen höherer Ordnung. Ziel sollte sein, Ausdrucksstärke unabhängig von Ausführung zu ermöglichen. In der Praxis orientiert sich Programmierer an weiteren Eigenschaften: Stil, Bibliotheken, Wartbarkeit, Besonders Performance

Eigenschaften stehen im Konflikt zu Deklarativer Programmierung. Funktionen höherer Ordnung auf normaler Hardware nur mit Overhead. Ebenso schränkt Endlichkeit von Computern Ausdrucksstärke ein. Folgen wie Fibonacci, Natürliche Zahlen, ... sind unendlich. Computer besitzen jedoch nur begrenzt Speicher.

In Haskell sollen beide Probleme gelöst werden. Beides durch die gleiche Technik, welche das Ausführungsmodell beschreibt. Dabei handelt es sich um STG [citation needed]

STG ist seit ??? [citation needed] wesentlicher Bestandteil von Haskell und im Übersetzungsprozess integriert. Im Ergebnis ist Laufzeit häufig vergleichbar mit C, jedoch auf weit höheren Abstraktionsebene.

1.1 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 Reduktionsschritte 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].

1.2 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.

2 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.

2.1 Lambda

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

Ansonsten wir gewohnt: Parameter Rumpf

2.2 Let-Bindings

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

Bedeutung ist Speichern auf dem Heap (Heap-Allokation)

2.3 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.

2.4 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.

2.5 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.

3 Implementierung

4 Ergebnisse

Abbildungsverzeichnis