Programmierparadigmen: Funktionale Programmierung

Thilo Gorfer

6. Juni 2013

Zusammenfassung

Ein kurzer Überblick über die funktionale Programmierung am Beispiel Haskell

Abstract

A short overview of functional programming with an example in Haskell

Inhaltsverzeichnis

1 Einleitung

In den folgenden Seiten möchte ich Ihnen einen kurzen Überblick über das Entstehen der Funktionalen Programmiersprachen und deren Vorteile geben. Später werde ich noch genauer auf deren syntaktische Eigenheiten am Beispiel von Haskell eingehen.

2 Der historische Hintergrund

Als noch kaum jemand an die Programmierung im heutigen Sinn mittels Höherer Programmiersprachen dachte, entstand die funktionale Programmierung. Sie findet ihren Ursprung in der Mathematik. Alonzo Church und Stephen Cole Kleene entwickelten 1932 das Lambda-Kalkül und 1936 wurde dessen Nutzen für die Informatik von Kleene und John Barkley Rosser nachgewiesen. Man konnte es nutzen um Funktionen in der Informatik umzusetzen. Das Lamda Kalkül wurde mit der Absicht entwickelt eine Sprache zu kreieren in der sich alle möglichen Probleme darstellen lassen [3]. Schließlich 1937 hat Turing bewiesen, dass das Lambda-Kalkül turing-äquivalent ist. Rein funktional geschriebene Programme können in Lambda Kalküle umgewandelt werden. 1960 erscheint LISP[1] als eine der ersten funktionalen Programmiersprachen. Entwickelt wurde sie von John McCarthy, seiner Zeit Professor am Dartmouth College. In seiner Erweiterbarkeit lag der große Vorteil von LISP. In den späten 70er Jahren erfand John Backus die Sprache FP – kurz für Function Programming. In FP setzte er seiner Idee von Funktionen höherer Ordnung (eine Funktion die entweder Funktionen als Parameter nimmt oder eine Funktion als Rückgabewert liefert) um. Zu ungefähr der gleichen Zeit entwickele man in Edinburgh and der Universität die Sprache ML mit der man unter anderem Theoreme beweisen konnte. Aus ML entstanden später weitere Dialekte wie zum Beispiel SML oder CAML.

[1] List Processing

3 Warum Funktionale Programmierung?

Objekt-orientierte und logische Programmiersprachen folgen dem imperativen Programmierparadigma. Sobald ein Programm ausgeführt wird arbeitet der Rechner die Befehle nacheinander ab. In diesen Sprachen stellen Variablen einen bestimmten Zustand dar, der sich während der Programmausführung ändern kann. Im Gegensatz dazu sind Variablen in funktionalen Sprachen unveränderlich und können somit nur für einen bestimmten Wert stehen. Diese unveränderlichen Variablen bieten einen großen Vorteil beim Multithreading und vereinfachen so den Code für parallel verarbeitete Programme erheblich. Ein weiterer wichtiger Punkt sind die Funktionen, welche ihre Eingabeparameter und auch den Zustand des Programmes nicht verändern dürfen. Somit kann eine Funktion nur ihren eigenen Zustand verändern, aber nicht den von anderen Bereichen des Programmes. Aus diesem Grund können keine Nebenwirkungen auftreten. Da in der Mathematik Funktionen sehr häufig verwendet werden, sind funktionale Programmiersprachen besonders in akademischen Kreisen weit verbreitet. [2]

4 Lambda-Kalkül

Das Lambda-Kalkül ist eine mathematische Schreibweise für Funktionen. Beispiel im Lambda-Kalkül:

$$mittelwert = \lambda a, b.((a + b)/2)$$

Beispiel in Haskell:

$$mittelwert(a, b) = ((a + b)/2)$$

Ein Vorteil des Lambda-Kalküls ist, dass bewiesen werden kann ob ein Programm korrekt ist oder nicht und da alle rein funktional geschriebenen Programme in Lambda-Kalküle umgewandelt werden können kann man deren Korrektheit sehr einfach beweisen. [3]

5 Haskell

5.1 Das Experiment

Die ARPA[2] und das Office of Naval starteten 1989 ein Projekt um eine geeignete Sprache für die Verbesserung der Erstellung von Prototypen zu finden. Im Zuge dieses Projekts wurde 1993 ein Versuch durchgeführt in welchem es darum ging, einen Prototyp in mehreren Programmiersprachen zu ein und demselben Problem zu schreiben. Der Prototyp sollte als Eingabe die Standorte von Schiffen, Flugzeugen und anderen Objekten erhalten und als Ausgabe Beziehungen zwischen diesen Objekten liefern.

Language	Lines of Code	Lines of Documentation	Development
Haskell	85	465	10h
Ada	767	714	23h
Ada9X	800	200	28h
C++	1105	130	-
Awk/Nawk	250	150	-
Rapide	157	0	54h
Griffin	251	0	34h
Proteus	293	79	26h
Relational Lisp	274	12	3h

Tabelle 1: Zusammenfassung der Prototypen

Aus der Tabelle 1 geht hervor, dass das Programm in Haskell kürzer war und wenig Arbeitsaufwand hatte. Das Komitee beurteilte den Code als sehr einfach und gut zu verstehen. Dadurch zeigt sich, dass Haskell sich sehr gut für Prototypenentwicklung eignet. [1]

[2]Advanced Research Project Agency

5.2 Syntaktik

5.2.1 Besonderheiten

In Haskell gibt es ein paar Besonderheiten. Da Semikolons am Ende von Anweisungen in anderen Programmiersprachen oft vergessen werden und somit zu Fehlern führen wurden diese in Haskell weggelassen. Aber sehr ungewöhnlich ist, dass in Haskell die Einrückung der Zeilen eine Rolle spielt und vom Compiler mit ausgewertet wird. Rückt man Zeilen nun falsch ein wird dies als Fehler gewertet oder kann zu falscher Semantik führen. Darauf sollte man vor allem bei if-Verschachtelungen achten. [2]

5.2.2 Typisierung

Haskell ist stark typisiert. Diese Typisierung erfolgt allerdings implizit durch Haskell und erspart dem Benutzer so eine explizite Typangabe. Im Englischen wird dies als type inference bezeichnet. Es bietet den großen Vorteil, dass schon zur compile-Zeit Fehler erkannt werden können. Zum Beispiel wenn in einer Funktion vor der Übergabe des Rückgabeparameters noch eine Division von 2 Integern stattfindet, so wird der Typ des Rückgabeparameters von Haskell auf Integer verfolgt und anhand dessen versucht im ganzen Programm weitere Typen festzulegen. [2]

5.2.3 Lazy Evaluation

Lazy evaluation bedeutet, dass Ausdrücke nur bei Bedarf ausgewertet werden und dies nur einmal. Beispiel:

$$f(g(x)) \tag{1}$$

f sei in diesem Beispiel eine Funktion die auf einer Liste arbeitet. g sei eine Funktion die Rekursiv eine unendlich lange Liste erzeugt und ein Element nach dem anderen der Funktion f übergibt. Würde hierbei keine lazy evaluation zum Einsatz kommen, würde die Funktion g unendlich lange laufen und das Programm steckt in einer Endlosschleife fest. Aber dank der lazy evaluation kommt es zu folgendem: Die Funktion f wird zuerst gestartet - wie auch ohne lazy evaluation - und fordert ein Listenelement von der Funktion g an. Dieser Wert wird nun von der Funktion f bearbeitet und darauf wird

wieder ein Element angefordert. Hier sieht man nun den Unterschied - erst wenn f einen neuen Wert anfordert erzeugt g diesen. Falls die Funktion f beendet wird sich auch g beenden und sofern f keine Endlosschleife erzeugt wird dieses Programm in endlicher Zeit ablaufen. Dank lazy evaluation muss man sich also in niedrigen Funktionen keine Gedanken machen wie lange zum Beispiel eine zu erzeugende Liste sein muss, da die höhere Funktion sich darum kümmern kann. Somit kann man niedrige Funktionen mehrfach wiederverwenden für höhere Funktionen die unterschiedlich viele Werte benötigen. Aus diesem Vorteil ergibt sich aber auch ein Nachteil, nämlich müssen sich höher geordnete Funktionen um das beenden der Funktion kümmern oder das Programm läuft endlos. Dass eine Auswertung nur einmal in einem Programm durchgeführt wird, bedeutet dass z.B. 7-3 nur einmal berechnet wird, selbst wenn diese Rechnung an den unterschiedlichsten Stellen im Programm geschieht. Hierdurch kann Rechenleistung gespart werden, allerdings erhöht das natürlich auch den Speicherverbrauch. [2]

Literatur

[1] Paul Hudak and Mark P. Jones. Haskell vs. ada vs. c++ vs. awk vs. ... Technical report, Yale Univesity Department of Computer Science, 1994.

[2] Graham Hutton. *Programming in Haskell*. Camebridge University Press, 2007.

[3] Christian Sternagel and Harald Zankl. Functional programming. *Skriptum zur Vorlesung Functional Programming*, pages 33–44, 2012.

Tabellenverzeichnis