Funktionale Programmierung

Zur Veranstaltung
Historische Paradigmenwechsel in der Informatik
im Sommersemester 2000
bei Prof. Siefkes
Tutoren:
Christian Siefkes
Joachim Korb

Vorgelegt von:
Aysel Alp
Yilmaz Uzun
Nezahat Alkan

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit soll der Frage nachgegangen werden, wie es zur Entwicklung des funktionalen Programmierparadigmas kam. Insbesondere wollen wir herausfinden, was die Beweggründe der Programmierer waren, was als Konzept vorgestellt wurde, ob und in wie weit sich die Ziele verändert haben. Im Folgenden wird versucht, die geschichtliche Entwicklung der funktionalen Programmierung nachzuvollziehen. Diese reicht von der Erfindung des Lambda – Kalküls im Jahre 1933 über Programmiersprachen wie LISP, ISWIM, FP, HOPE, MIRANDA, HASKELL bis hin zu OPAL im Jahre 1986/87. Es soll versucht werden, die diesen Programmiersprachen gemeinsame Idee herauszuarbeiten.

Als besonders wichtig erscheint uns in diesem Zusammenhang der Text von John Backus „ Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs“ [Backus,1978]. John Backus, einer der bedeutendsten Vertreter der funktionalen Programmierung, hat sich in diesem Text unter anderem auch ausführlich mit den Vorteilen der funktionalen Programmierung im Gegensatz zu den herkömmlichen (imperativen) Programmiersprachen auseinandergesetzt. Es bleibt zu erwähnen, daß er selbst einer der Mitbegründer der imperativen Programmiersprachen war. John Backus war an der Entwicklung der imperativen Programmiersprache FORTRAN beteiligt.

Ein weiterer Vertreter der funktionalen Programmiersprache ist Peter Pepper, der die funktionale Programmiersprache OPAL entwickelt hat.
Laut Peter Pepper ist das zentrale Anliegen der funktionalen Programmierung, etwas von der „Eleganz, Klarheit, Präzision“ der Mathematik in die Welt der Programmierung einfließen zu lassen.[1]

[1] Funktionale Programmierung , [Pepper 1986/87, S.2]

Hauptteil

2 Was versteht man unter funktionaler Programmierung ?

Die funktionale Programmierung beruht, wie der Name schon verrät, auf Mathematik und Logik. Die Programme werden nicht konkret auf Maschinen bezogen, sondern auf einem abstrakten, mathematischen Niveau formuliert. Sie sind im Gegensatz zu den imperativen Sprachen eine Ein-/Ausgaberelation, die im Programmtext direkt als Funktionen geschrieben werden. Die Programme sind „zeitlos“, hängen also nicht vom aktuellem Zustand der Maschine ab.

3 Warum funktionale Programmierung?

Die funktionale Programmierung war lange Zeit wegen der Beweisbarkeit, Klarheit usw. in akademischen Kreisen hoch angesehen. Allerdings war ihre Umsetzung auf existierenden Computern extrem ineffizient. Einer der Gründe, warum funktionale Programmierung inzwischen angewandt wird, ist in neueren Entwicklungen im Hardwarebereich zu suchen. Früher war die Herstellung von Prozessoren sehr teuer, was nicht mehr in so hohem Maße der Fall ist.
Um die Rechenleistung eines Computersystems zu erhöhen, setzt man unter anderem auch parallel geschaltete Prozessoren ein. Um die Möglichkeiten der parallel geschalteten Prozessoren voll ausschöpfen zu können braucht man eine Programmiersprache, die nicht sequentiell aufgebaut ist, sondern so, daß die zu lösenden komplexen Problemstellungen so beschrieben werden, daß eine gleichzeitige Lösung der Problemstellungen möglich ist. Funktionale Programmiersprachen sind für den Einsatz von solchen parallel geschalteten Prozessoren sehr geeignet.
Ein anderer Grund ist der Wunsch Spezifikationen für die Programme präziser zu machen, um die Entsprechung zwischen dem Programm und seiner Spezifikation besser in den Griff zu bekommen. Dieser Wunsch beruhte auf der Erkenntnis, daß etwa 50 % der Arbeitsanstrengungen eines Programmierers in die Behebung von oft tiefsitzenden, schwer auffindbaren Fehlern der Programme gehen. Aus diesem Grund war man auf der Suche nach einer eindeutigen Programmiersprache für die Spezifikation. Dabei bot sich eine aus der Mathematik stammende Programmiersprache als beste Lösung an. Auf diese Art kann dann der

Zusammenhang zwischen Programm und Spezifikation mathematisch bewiesen werden. Ein großes Hindernis für den Einsatz mathematischer Methoden beim Programmieren ist der Gebrauch der Variable als ein benannter Speicherplatz, dessen Inhalt über Zuweisungen verändert werden kann. Um sagen zu können, wofür eine solche Variable steht, muß man wissen, an welchem Punkt des Programmablaufs man sich befindet. Eine mathematische Variable dagegen hat einen Wert. Wenn dieser Wert noch nicht berechnet wurde, ist diese Variable unbekannt und hat nicht irgendeinen anderen Wert.
In den funktionalen Programmiersprachen haben solche Variablen diese, unter dem Namen referenzielle Transparenz bekannte, Eigenschaft. Diese hat neben der mathematischen Beweisbarkeit noch einen weiteren Vorteil. Da die unbekannten Variablen eines Ausdrucks ganz einfach nicht ausgewertete Funktionsaufrufe sind, die erst dann bekannt werden, wenn die Funktion ausgewertet wird, wird der Unterschied zwischen Funktion (Code) und Variablen (Daten) aufgehoben. Dies führt zu Funktionen höherer Ordnung und somit auch dazu, daß der Programmierer Funktionen und Daten mit gleicher Leichtigkeit manipulieren und strukturieren kann.

Ein funktionales Programm entspricht im wesentlichen einer Funktion im mathematischen Sinne, die auf die Eingabewerte angewendet wird und den Funktionswert als Ausgabe liefert. Der Lambda-Kalkül von Church und seine Theorie bilden die mathematische Basis der funktionale Programmierung. Auf Grund dieser mathematischen Tradition haben funktionale Programmiersprachen eine verhältnismäßig einfache Semantik, die eine wichtige Grundlage für Korrektheitsbeweise bildet. In diesem Zusammenhang ist auch die Eigenschaft der „referential transparency“ zu nennen: Der Wert eines Ausdruckes hängt nur von seiner Umgebung, nicht aber vom Zeitpunkt seiner Auswertung ab. In einem funktionalem Programm kann ein Teilausdruck also immer durch einen anderen Ausdruck der den selben Wert hat, ersetzt werden. Die Semantik des Programms bleibt unverändert. Der Programmentwurf erfolgt in einer funktionalen Sprache auf einer höheren Abstraktionsstufe als in einer imperativen Sprache.

4 Einiges zur Funktionalen Programmierung

Der Lambda-Kalkül von Church wird als Ursprung und gemeinsamer Kern aller funktionalen Sprachen angesehen. Churchs Intention bei der Entwicklung des Lambda-Kalküls war eine „Formalisierung des Berechenbarkeitsbegriffs auf der Basis von Funktionen“. Sein Ziel war vor allem die Erfassung der Grundaspekte der Berechnung von Funktionen in einem möglichst formalen Kalkül. Dies führte zu der Berühmten “ Churchschen These“ der Berechenbarkeitstheorie: alle intuitiv berechenbaren Funktionen sind im Lambda-Kalkül definierbar.
Die erste Funktionale Sprache LISP (**LIS**T **P**ROCESSING) wurde Anfang der 60er Jahre von John McCarthy entwickelt. LISP war die erste Sprache, die weit von den Prinzipien der damaligen, durch die Sprache FORTRAN geprägten Programmierung abwich. Eine der bedeutendsten Ansprachen für die funktionale Programmierung und gegen den imperativen Programmierstil hielt John Backus 1977 anläßlich der Verleihung des „Turing Award“ an ihn. Er machte die imperative Programmierung für die Software-Krise verantwortlich und schlug als Alternative eine funktionale Sprache vor.
Seine Rede löste eine Flut von Neuentwicklungen aus. Großen Einfluß auf die Entwicklung funktionaler Programmiersprachen übte David Turner aus. Als Resultat einer Folge von Sprachentwicklungen stellte er 1985 Miranda vor. Miranda zeichnet sich durch die Verwendung von bedingten rekursiven Gleichungen zur Definitionen von Funktionen und Funktionen höherer Ordnung aus.

5 Kritik von John Backus an den konventionellen Programmiersprachen

In seiner Einleitung kritisiert John Backus die damals konventionellen Programmiersprachen als „...das Reich derer, die sich mit Anhäufungen von Details auseinandersetzen, anstatt um neue Ideen zu ringen“. Ihm erscheinen die Diskussionen über Programmiersprachen „ ...wie mittelalterliche Debatten über die Anzahl der Engel, die auf einer Nadelspitze tanzen können.“ [2]
Man verbeiße sich in unsinnige Details, anstatt über grundsätzlich unterschiedliche und neue Konzepte zu streiten. Zwar hätten viele kreative Informatiker „elegante“ neue Methoden entwickelt, die sie jedoch auf die Analyse der konventionellen Programmiersprachen „verschwendet“ hätten, anstatt ihre Energie auf die Erfindung von etwas Neuem zu verwenden.
Beim Lesen dieses Textes entsteht leicht der Eindruck, daß Backus, der wie viele Vertreter der Funktionalen Programmierung, ursprünglich Mathematiker war, durch die Formen, die sich in der konventionellen Programmierung entwickelt hatten, in seinem ästhetischen Empfinden beleidigt fühlte. Der Ironische Stil im Abschnitt „Conventional Programming Languages: Fat and Flabby“ läßt darauf schließen. Ein konkretes Beispiel dafür ist der schon oben aufgeführte Satz „ ...wie mittelalterliche Debatten über die Anzahl der Engel, die auf einer Nadelspitze tanzen können.“
Nach John Backus beruht jede Programmiersprache auf einem Modell eines Computersystems. Für die funktionale Programmiersprachen entwarf er ein Modell, welches er nach seinen Vorstellungen entsprechend in Kriterien unterteilte.

[2] [Backus 1978, S.614]: „ Discussion about programming languages often resemble medieval debates about the number of angels that can dance on the head of a pin instead of exciting contest between fundamentally differing concepts.“

Bevor wir die Kriterien einzeln auflisten, möchten wir darauf eingehen, wie es zu solch einem Modell kommt. Als erstes muß der Entwickler eine Vorstellung davon haben, was für ihn der Computer darstellt. Dieser Vorstellung entsprechend entwickelt er dann eine Programmiersprache.
John Backus klärt uns nicht darüber auf , was er unter den Grundlagen eines Modells versteht. Es ist aber herauszulesen, daß es ihm im Wesentlichen darum geht, daß die Grundlagen des Modells mathematisch beweisbar sind.

Dies waren die Kriterien für das Modell von John Backus für eine funktionale Programmiersprache:

1. Grundlagen: Hierbei ging es ihm darum, ob es eine elegante, kurze mathematische Beschreibung existiert.
2. History sensitivity: Dies bedeutet, ob das Programm Informationen speichern kann, welche die nachfolgenden Programme beeinflussen.
3. Semantik: Sind Zustände einfach oder kompliziert, d.h.: Ist ein Programm so geschrieben, daß man den Wert einer Variablen auf Anhieb bzw. nach kurzem Nachvollziehen erkennt oder nicht.
4. Klarheit: Sind die Programme mit ihren Befehlen ein klarer Ausdruck des Rechenprozesses? Gibt es eindeutige Begriffe innerhalb des Programms, die uns helfen, über den Rechenprozeß zu argumentieren?

Die konventionellen Programmiersprachen können zwar einigermaßen klar sein, jedoch sind sie von der begrifflichen Seite her nicht zu gebrauchen.
Konventionelle Programmiersprachen orientieren sich am „von Neumann Computer“, der, in den Worten von John Backus, um ein Flaschenhals herumgebaut ist. Die CPU und der Speicher sind durch eine Leitung miteinander verbunden. Die CPU und der Speicher tauschen ihre Daten miteinander mit Hilfe dieser Leitung, der ein „schmales Rohr“ zwischen beiden bildet, da es nur eine Informationseinheit pro Zeiteinheit durchläßt. Dies beeinflußt natürlich den Programmierstil. Ein weiteres Problem der „von Neumann Computer“ liegt darin, daß der Aufbau des Programms zwar klar sein kann , die Begriffe innerhalb des Programms jedoch nicht eindeutig bestimmt sind.

Ein weiterer Kritikpunkt ist die Unflexibilität der „von Neumann Sprachen“. Sie enthalten nur wenige vom Benutzer veränderliche Teile. Das ist auch darauf zurückzuführen, daß ihre Semantik zu eng an den Zustand, der sich in jedem Rechenschritt verändert, gekoppelt ist. Die vom Benutzer veränderbaren Teile besitzen weiterhin eine nur geringe Ausdruckskraft (dies führt dazu, daß der in der Programmiersprache fest vorgegebene Rahmen sehr große Dimensionen annimmt).

Darüber hinaus besteht ein wichtiger Mangel der „von Neumann Sprachen“ darin, daß ihnen nützliche mathematische Eigenschaften fehlen, welche für das Argumentieren über Programme nützlich wären. Für John Backus gibt es keine Alternative für das Beweisen bzw. Argumentieren von Programmen außer der Mathematik.

John Backus kommt zu dem Schluß, daß Beweise in jedem Fall nur mittels der Sprache der Logik über Programme argumentieren, diese jedoch nicht direkt miteinbeziehen können. Seine Idee besteht darin, daß eine Programmiersprache eine ihr assoziierte Algebra, also im Falle der funktionalen Programmiersprachen Rechenregeln für Funktionale (höhere Funktionen) und Funktionen, besitzen sollte, so daß mit den Programmen ähnlich wie beim Auflösen einer Gleichung algebraisch bewiesen werden kann.
John Backus hat diesen Vortrag „Can Programming Be Liberated from the von Neumann Style?“ 1978 gehalten. In den 80`er Jahren entstand eine Reihe von funktionalen Programmiersprachen, die von HOPE bis zu HASKELL, GOFER und OPAL reichen. Die in dem Text von Backus skizzierten Grundkonzepte stimmen jedoch bei all diesen Sprachen überein. Es wird von den Vertretern der Funktionalen Richtung behauptet, daß die Funktionalen Programmiersprachen tatsächlich einen klaren eleganten Programmierstiel ermöglichen und darüber hinaus die Beweisbarkeit der Programme erhöhen.
In einer Funktionalen Programmiersprache kann man leichter und schneller schreiben. Für Anfänger ist jedoch die funktionale Programmiersprache, die sehr viele rekursive Programme enthält, sehr viel schwerer nachzuvollziehen, als zum Beispiel ein PASCAL Programm.

6 Gegenüberstellung der konventionellen und funktionalen Sprachen

Das Spektrum der Programmiersprachen erstreckt sich von dem imperativen Programmiersprachen, in denen Programme im wesentlichen Abfolgen von Befehlen an Rechner beschreiben, bis zu den funktionalen Programmiersprachen, in denen Programme eher Problembeschreibungen als Ausführungsanweisungen an einem Rechner sind. In den imperativen Sprachen, zu denen Sprachen wie FORTRAN, PASCAL etc. zählen, beschreiben Programme Berechnungen als Serie von lokalen Speichertransformationen eines von Neumann Rechners. Beispielsweise bewirkt die Wertzuweisung X:= X+3 die lokale Modifikation des Inhaltes der Speicherzelle, die der Variablen X zugeordnet ist. Die Programmierung erfolgt rechnernah, was auf die evolutionäre Entwicklung dieser Sprachen aus rechnernahen zurückzuführen ist. Also ist ein Programm in der imperativen Programmiersprache eine Arbeitsanweisung für eine Maschine. Was das Programm tut, hängt vom Zustand der Maschine ab. Diese ändert sich im Laufe der Zeit an die der Programmierer denken muß, wenn er sein Programm verstehen will.

Funktionale Programmiersprachen wie LISP, MIRANDA etc. zeichnen sich demgegenüber dadurch aus, daß sie auf einer rein mathematischen Theorie fundieren. Die Programmierung erfolgt in diesen Sprachen rechnerunabhängig auf einem höheren Abstraktionsniveau. In der funktionalen Programmiersprachen ist ein Programm eine Ein-/ Ausgaberelation, d.h., sie ist eine Abbildung von Eingabedaten auf zugehörige Ausgabedaten. Das in der funktionalen Programmiersprache geschriebene Programm liefert zu jedem Zeitpunkt ein und das selbe Ergebnis und ist somit „zeitlos“.

7 Unterschied zwischen konventionellen und funktionalen Programmiersprachen als Tabelle

Konventionell	funktional
Programme werden maschinenbezogen formuliert	Aufgebaut auf Mathematik und Logik
Programme sind eine Arbeitsanweisung für eine Maschine	Programme sind eine Ein -/ Ausgaberelation, die im Programmtext direkt als Funktion hingeschrieben wird
Was ein Programm tut, hängt vom Zustand der Maschine ab, daher muß der Zeitaufwand mitberechnet werden	Programme sind zeitlos, d.h., das Programm hat zu jedem Zeitpunkt ein und das selbe Ergebnis.
history sensitivity	Wenig history sensitivity
Keine assoziierte Algebra	Assoziierte algebraische Eigenschaften

8 Schluß

Der Kernunterschied zwischen imperativen und funktionalen Programmiersprachen ist, wie der Name schon sagt, daß in den funktionalen Programmiersprachen jedes Programm eine Funktion darstellt. Dies wird auch in der Tabelle näher gezeigt (siehe Seite 9). Dies ist auch der Ansatz von John Backus. Er versieht diese „Funktionswelt“ mit einer assoziierten Algebra, d.h.: Funktionale, Funktionen und ihre Verknüpfungen genügen gewissen einfachen mathematischen Rechenregeln. Auf diese Art wird das Verhalten eines Programms ohne Bezug auf die Maschine mathematisch analysierbar. Diese Vorstellung von John Backus kommt unserer Ansicht nach in den in der Folgezeit entwickelten funktionalen Sprachen nur wenig zur Geltung.
Die funktionalen Programmiersprachen sind wenig populär. Die auch relativ neu entwickelten objektorientierten Programmiersprachen, wie zum Beispiel C++, JAVA stehen zur Zeit sehr im Trend. Der Grund dafür liegt unserer Meinung nach in dem hohen, abstrakten, mathematischen Niveau, auf dem die funktionalen Programmiersprachen formuliert sind, was wir auch beim Lesen des Textes von John Backus feststellen konnten.

Literatur

John Backus	„Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs [ACM Turing Award Lecture, 1978]
Susan Eisenbach	Functional Programming Languages, Tools and Architectures [1987]
Peter Pepper	Funktionale Programmierung in OPAL, ML, HASKELL und GOFER [1986/87]