Funktionale Programmierung I, Sommersemester 2000

Prof. Dr. Manfred Schmidt-Schauß

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
E-mail:schauss@ki.informatik.uni-frankfurt.de

Zeiten: Vorlesung: Di,Do: 10:00-12:00 SR 11

Scheinerwerb: Übungsaufgaben und Teilnahme an Übungen

Verwendete Programmiersprachen: (auf HP, PC, .. verfügbar)

- Gofer
- Hugs (Haskell)

Literatur:

Bird, R., Introduction to Functional Programming using Haskell, Prentice Hall, (1998) (sehr gute Einführung in alle Themen) ISBN: 0-13-484346-0 (bei amazon.uk, ca. DM 100)

Bird, R., Wadler, Ph., Introduction to Functional Programming, Prentice Hall, (1988) (sehr gute Einführung in alle Themen, Programmiersprache Miranda)

Thiemann, Peter Grundlagen der funktionalen Programmierung, Teubner Verlag, (1991) (empfohlene Einführung in alle Themen, Programmiersprache Haskell)

Davie, J. An introduction into functional programming using Haskell, Cambridge University Press, (1992) (Eine Einführung in Haskell und andere Themen mit verständlichen Beispielen)

Hinze, Ralf , Einführung in die funktionale Programmiersprache Miranda, Teubner Verlag, (1992)

Jeuring, J., Meijer. E., eds. Advanced functional programming, Lecture Notes in Computer Science 925, Springer-Verlag (1995) (Ein Kurs und verschiedene Anwendungen)

Abelson, H. Sussmann,G.J. Structure and Interpretation of Programs, MIT Press, (1985) (Einführung in Informatik anhand der Lisp-Variante Scheme) (gibt es auch in deutsch)

Thompson, Simon, Miranda, The craft of functional programming, Addison Wesley, (1995) (Miranda-Einführung

Plasmeijer, R., von Eekelen, M., Functional Programming and Parallel Graph Rewriting, Addison Wesley, (1993) (Funktionale Programmierung aus der Sicht der Gruppe, die Clean entwickelt)

Runciman, C. Wakeling, C. , Applications of Functional Programming, UCL Press London, (1995) (kurz gehaltene Einführung, ausführliche Diskussion von Anwendungsproblemen in Haskell

Weiterführende Literatur:

Barendregt, H.P. The Lambda Calculus: - Its Syntax and Semantics, North Holland, (1984) (Theorie des Lambda-Kalküls)

Peyton Jones, S. The Implementation of Functional Programming Languages, Prentice Hall, (1987) (Standardwerk zur Implementierung von verzögert auswertenden funktionalen Programmiersprachen)

Peyton Jones, S., Lester,D., Implementing Functional Languages, Prentice Hall, 1991 (Begleitbuch zu einem Praktikum)

Richard Bird, Oege de Moor Algebra of Programming] Behandelt Programmieren und Programmtransformation auf einer abstrakteren Ebene. Ist mit Haskell zusammen verwendbar.

Davey, B.A., Priestley, H.A., Introduction to Lattices and Order, Cambridge University Press, (1990) (Ordnungen, Verbände, usw: geeignet für Betrachtungen zur Semantik)

Huet, G., Logical foundations of functional programming, Addison Wesley, 1990

ODonnel, M. , Equational Logic as a programming language, MIT Press, 1986

Sleep, M.R., M.J.Plasmeijer, M.J., Eekelen, M.C.J.D. , Term graph rewriting, Wiley, 1993

Kluge, W., The organization of reduction, data flow, and control flow systems, MIT Press, (1992)

Steele, Guy L. COMMON LISP, THE LANGUAGE, second edition, digital press (1990)

Handbuch-Artikel:

Odersky Funktionale Programmierung, Kapitel D5 in Rechenberg, Pomberger: Informatik Handbuch, Hanser-Verlag

Klop: Term Rewriting Systems, Handbook of Logic and Computer Science

Barendregt, H.P. functional programming and the Lambda-calculus, Handbook of Theoretical Computer Science, Vol B, Elsevier, (1990)

Mitchell, J.C. , Type systems for programming languages, Handbook of Theoretical Computer Science, Vol B, Elsevier, (1990)

Klassische Artikel:

Backus: Can programming be liberated from the von Neumann style, Comm ACM 21, 218-227, (1978)

Turner, D.A. , A new implementation technique for applicative languages, Software Practice & Experience 9, pp. 31-49, (1979)

Turner, D.A., Miranda, a non-strict functional programming language with polymorphic types Proc. Conf. on Programming Languages and Computer Architecture, LNCS 201, Springer Verlag, (1985)

Milner,R. Theory of type polymorphism in programming. J. of Computer and System Sciences 17, pp. 348-375, (1978)

Inhalt (voraussichtlich):

– Einführung
– Überblick aktuelle funktionale Programmiersprachen
– Kernsprache KFP: Terme, Syntax, operationale Semantik und Auswertungsstrategien
– Übersetzung in den Lambda-Kalkül Normalformen, WHNF, ... Fixpunkte,
– Einführung in Sprachen der Lisp-Familie: Scheme, ML, Common-Lisp
– Polymorphe Typsysteme
– Einführung in Gofer und Haskell Typklassen
– Programmiertechniken + Datenstrukturen Rekursion, Iteration, Modularisierung, Listen: map, filter, fold, ... List Comprehensions Bäume, Graphen, ...
– Kombinatoren, List-of-Successes (parsing ...)
– monadisches Programmieren
– Kompilierung + Implementierungsmethoden Transformation von Programmen Graph-reduktion, G-Maschine, Speicherverwaltung, garbage collection STG-Maschine
– Analysetechniken
– induktive Verifikationsmethoden

Ziel:

– funktionale Programmiersprachen + zugehörige Konzepte verstehen.
– Vor / Nachteile der Konzepte
– Betonung auf nicht-strikten (verzögert auswertenden)

1 Einführung

Man kann Programmiersprachen grob danach unterteilen, ob sie imperativ oder deklarativ sind. Imperative Programmiersprachen sind dadurch gekennzeichent

dass eine Folge von Anweisungen bearbeitet wird, bei denen es auf den Gesamteffekte auf die Umgebung (Hauptspeicher, Drucker, Datenbank, Bildschirm, usw.) ankommt. D.h. es wird beim Programmieren geaagt WIE ein Problem bzw. Aufgabe zu lösen ist.
Bei deklarativen Programmiersprachen wird vom WIE weitgehend abstrahiert und stattdessen eine Beschreibung des gewünschten Ergebnisses durchgeführt. Die Berechnung des Ergebnisses ist nicht explizit in der Bescrheibung enthalten. Z.B. sind Spezifikationen in verschiedener Form eher deklarativ. Die logischen und die funktionalen Programmiersprachen rechnet man zu den deklarativen Programmiersprachen.

Funktionales Programmieren **Programmieren** = Definition von Funktionen

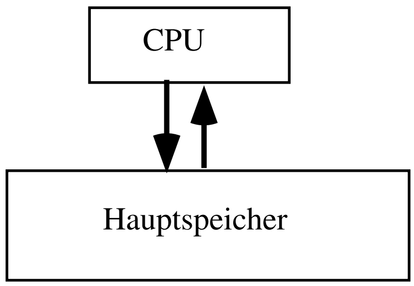
Ausführung = Auswerten von Ausdrücken

Resultat: Einziges Resultat eines Ausdrucks oder eines Programms ist der Rückgabewert bzw. der Wert des Ausdrucks.

$$Resultat = Funktion(Argument_1, \ldots, Argument_n)$$

Wichtig:! Es gibt (i.a.) keinen expliziten globalen Zustand, der von den Funktionen verändert werden kann. (Nicht-von-Neumann)

von-Neumann Programmieridee: Schema der von-Neumann Architektur: (vereinfacht)



Programm = Folge von Anweisungen zum Verändern von Werten im Hauptspeicher

Auswertung: Folge von Zuweisungen, Abfrage von Bedingungen, und Verändern von Werten und/oder Programmzähler

Daten und kompiliertes Programm sind im gleichen Hauptspeicher.

Imperative Programmiersprachen:

Folge von Anweisungen manipuliert den Zustand (Speicherinhalt) und die Steuerung (Programmzähler)

Programmiersprachen: Assembler, Fortran, Cobol Algol, Pascal, C, C++ , ...

Objektorientierte Programmiersprachen:

Klassen, Methoden, Vererbung, Objekte,

Wesentliche Neuerung: Strukturierung des Programms und der Daten durch Angabe von Klassen,

von-Neumann Programmieridee ist Teil des Konzeptes: auch hier gibt es einen Zustand, der manipuliert wird.

Logisches Programmieren: (Prolog)

Basiert auf einer Menge von prädikatenlogischen Formeln: Fakten und Regeln (Hornklauseln) als Programm.

Die Ausführung entspricht einer iterierten Folge von logischen Schlüssen, sogenannte Resolution.

Funktionales Programmierung

Prinzip:

- Gleiche Funktion angewendet auf gleiche Argumente ergibt gleichen Wert
- Die Funktion f verändert sich nicht durch die Verarbeitung
- Sie darf sich auch nichts heimlich irgendwo merken. Nur die Argumente bestimmen den Wert.

Referentielle Transparenz: Gleiche Argumente ergeben gleichen Wert.
Variablen bezeichnen Werte, nicht Speicherplätze; Ausdrücke bezeichnen Werte
Substitutionsprinzip: Im Programmtext können Ausdrücke mit gleichem Wert ausgetauscht werden ohne Konsequenz für den Wert des Gesamtausdrucks.
Wichtig ist der **Wert des Ausdrucks**, nicht die Reihenfolge der Auswertung
WAS nicht WIE

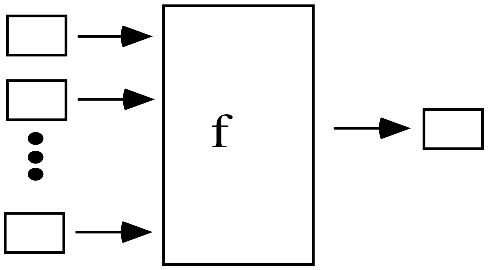
Kommentar zu objektorientierte Sprachen

Funktionale Programmiersprachen liegen etwas quer zu einigen unter dem Stichwort OO versammelten Programmierprinzipien:
Modularisierung und information-hiding ist in funktionalen Sprachen zum Exzess getrieben. Referentielle Transparenz macht die Wiederverwendung von Funktionalität leicht möglich. Strenge Typisierung gibt dem Programmierer (und Anwender) die Sicherheit, daß eine großer Teil der möglichen Fehler ausgeschlossen ist. Typklassen in Haskell mit Vererbung sind sehr analog zum Klassensystem der OO-Sprachen. Ausdrücke haben einen Typ; der Typ gehört zu einer Klasse abhängig von der Klasse werden andere Funktionen (Methoden) benutzt

Die folgenden von-Neumann-Anteile fehlen in reinen funktionalen Sprachen: Objekt-Identität + Veränderbarkeit von Objekten

Java Hat Klassen und Methoden Ausdrücke mit Wert
Es gibt Zuweisungen
automatische Speicherverwaltung
keine Pointer-arithmetik

Denotationale Semantik von (imperativen und funktionalen) Programmiersprachen :
Bildet Anweisungen bzw. Funktionen ab auf Ausdrücke des Lambda-Kalküls (bzw. Ausdrücke rekursionsfreier funktionaler Programmiersprachen).



"Programm"	"Bedeutung"
Zeichenketten	mathematische Objekte (denotationale Semantik)
	formale Beschreibung der Wirkung (operationale Semantik)
$(s\ 0) + s\ (s\ 0)$	$1 + 2$
$q\ x := x * x$	q wird abgebildet auf die Quadratfunktion über Zahlen
$f\ x := x\ +\ f\ x$	undefiniert (terminiert nicht)

Funktions/Prozedur-aufrufe in Programmiersprachen:
Seiteneffekt: Änderung des Speichers (der Programmierumgebung) während eines Funktions/ Prozeduraufrufs, wobei diese Änderung nach dem Aufruf noch Auswirkungen hat.

1.1 Prozeduraufruf: Methoden bei imperativen Programmiersprachen

Methode 1) (call by name)

Funktion bekommt n Argumente (als Pointer bzw. Speicherplätze) der Wert ist an einer bekannten Adresse.
(Seiteneffekte sind notwendig)

Methode 2) (call by value)

Funktion erhält nur die Argumentwerte, Speicherzugriff möglich
Seiteneffekte sind nicht notwendig, aber i.a. möglich durch Veränderung von globalen Variablen

Pascal, Lisp, Scheme, ML, ...

Methode 3) (call by value,seiteneffektfrei)
Funktion erhält nur die Argumentwerte, liefert Wert zurück

keine Seiteneffekte
Haskell, pures Prolog, pures Scheme, pures Lisp, .. pur = seiteneffektfrei

Seiteneffekte:

Vorteile: unterstützt von Hardware, effizient, Übergabe von Werten ist einfach
Nachteile: Anweisungen mit Seiteneffekt erzwingen (bzw. erfordern) Sequentialisierung durch den Programmierer. verhindern bzw. erschweren Parallelisierung.

- Die Semantik der Funktion ist dann nur erklärbar durch Argument / Wert Zuordnung für gegebenen globalen Zustand und Veränderung des globalen Zustandes
- eine definierte Funktion / Prozedur hat nicht lokale Eigenschaften
- Verifikation von Prozeduren ist schwierig und komplex,
- Verhalten der Prozedur ist nicht nur von Argumenten, sondern vom aktuellem Speicherinhalt (der aktuellen Umgebung) beeinflußt.
- Die Wiederverwendbarkeit von Funktionen/Prozeduren ist sehr erschwert

1.2 Aktuelle funktionale Programmiersprachen

Unterscheidungskriterien:

Typsystem: **statisch:** zur Compilezeit
dynamisch: Test zur Laufzeit
stark: Typfehler werden vom Compiler nicht toleriert, alle Ausdrücke sind getypt
schwach: Typfehler werden vom Compiler toleriert, manche Ausdrücke sind ungetypt
monomorph: Funktionen haben einen festen Typ
polymorph: Funktionen haben schematischen Typ (Typvariablen)
Überladung: Funktionen können so definiert werden, daß sie für mehrere (fest vorgegeben) Typen verwendbar sind (nicht schematisch) (analog zu OO-Methoden)
Typklassen: Typen sind Klassen zugeordnet

Status der Objekte: erster Ordnung: Argumente/Werte können nur Datenobjekte sein höherer Ordnung: Funktionen sind als Objekte zugelassen können Argument/Wert von Funktionen sein

Auswertung strikt: Argumente werden bei Übergabe ausgewertet nicht-strikt (lazy, verzögert): nicht ausgewertete Argumente möglich

Speicherverwaltung automatisch / “garbage collector“

Argumentübergabe mittels Selektoren / pattern matching (keine Selektoren notwendig)

Klassifikation nach Funktionsaufrufmethode

Lisp Common-Lisp Standard: Allegro-Common-Lisp CLOS: Common Lisp Object system ungetypt; strikt, Funktionen höherer Ordnung; Seiteneffekte, patterns teilweise möglich

Scheme: Lisp-Variante: ungetypt; strikt, Fun. höherer Ordnung; Seiteneffekte, keine patterns

ML: Standard ML strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; Seiteneffekte, patterns

ID lazy, ungetypt; erster Ordnung; Seiteneffekte

Erlang strikt, ungetypt, eingeschränkt höherer Ordnung, funktionale Sprache von Ericsson, Seiteneffekte Anwendung in Telekommunikationsanlagen.

Sisal: monomorph getypt, strikt, erster Ordnung, implizite Parallelisierung gemeinsamer Speicher.

HOPE strikt statisch stark getypt; Fun. erster Ordnung; patterns, Seiteneffeke?

LML (Lazy-ML) nicht-strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; patterns, keine Seiteneffekte

Miranda nicht-strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; patterns, keine Seiteneffekte,

NEL nicht-strikt, polymorph, statisch stark getypt; Fun. höherer Ordnung; patterns, eingeschränkte Seiteneffekte, industrielle Anwendungen

Gofer / Haskell nicht-strikt, polymorph statisch stark getypt + Typklassen, patterns, Funktionen höherer Ordnung; keine Seiteneffekte

CLEAN: nicht-strikt, polymorph statisch stark getypt; Funktionen höherer Ordnung; patterns, keine Seiteneffekte

1.3 Einordnung von imperativen Programmiersprachen

C, Pascal monomorph bzw. schwach getypt, teilweise statisch; teilweise strikt, erster Ordnung; Seiteneffekte keine automatische Speicherverwaltung, keine patterns

Java Typsystem: dynamisch getypt, teilweise statisch; strikt, erster Ordnung; Seiteneffekte automatische Speicherverwaltung