

Einführung in die funktionale Programmierung,  
Wintersemester 2007/08

Prof. Dr. Manfred Schmidt-Schauß  
Fachbereich Informatik und Mathematik  
Johann Wolfgang Goethe-Universität  
Postfach 11 19 32  
D-60054 Frankfurt  
E-mail:schauss@ki.informatik.uni-frankfurt.de

15. Oktober 2007

# Kapitel 1

## Einführung

### 1.1 Allgemeines

Die **Einführung in die funktionale Programmierung** findet in der ersten Semesterhälfte statt, die Fortsetzung (für Diplom-Informatiker) in der zweiten Semesterhälfte ist die Vorlesung:

**Funktionale Programmierung: Analyse von Programmen.**

Sehr viele Informationen, Ressourcen und Links zu dieser Vorlesung incl. diesem Skript befinden sich auf der Web-Seite der Professur <http://www.ki.informatik.uni-frankfurt.de/lehre/WS2007/main.html> bzw. im univIS-System der Universität Frankfurt oder auf den Web-Seiten des Instituts. Die Bedingungen zum Erwerb eines Leistungsscheins (Diplom) bzw. zum erfolgreichen Abschluss des Bachelor-Moduls sind ebenfalls dort zu finden.

### 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. 50 Euro)

**Manuel M. T. Chakravarty und Gabriele C. Keller** : Einführung in die Programmierung mit Haskell, Pearson Studium (2004)  
Verständliche Einführung in die Programmierung in Haskell

**Peter Pepper und Petra Hofstedt** Funktionale Programmierung (2006)  
Grundlagen und Konzepte zu strikten und nicht-strikten funktionalen Programmiersprachen. Haskell, ML und Opal werden parallel besprochen

**Hankin, C.** An Introduction to Lambda Calculi for Computer Scientists, King's College Publications, (2004)

**Simon Thompson** : Haskell: The Craft of Functional Programming

**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)

#### **Weiterführende Literatur:**

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

**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)

**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

**ODonnell, 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)

### 1.1.1 Inhalt (für Bachelor und Diplom)

- Einführung in Haskell
- operationale und kontextuelle Semantik; Haskell, Kernsprachen KFP, KFPT, KFPTS
- monadisches Programmieren
- Typsysteme: parametrischer Polymorphismus, Typklassen
- Programmier Techniken + Datenstrukturen  
Rekursion, Iteration, Modularisierung, Listen: map, filter, fold, ...  
List Comprehensions, Bäume, Graphen, ....
- Typberechnung, Verfahren nach und iteratives Verfahren.

### Inhalte des zweiten Teils: (Diplom Informatik)

- Operationale Semantik und kontextuelle Gleichheit
- Übersetzung in den Lambda-Kalkül  
Normalformen, WHNF, ... Fixpunkte,
- Programmtransformationen und deren Korrektheit
- Induktive und co-induktive Beweis- und Verifikationsmethoden

## 1.2 Einführung

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

Imperative Programmiersprachen sind dadurch gekennzeichnet, 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 gesagt WIE ein Problem bzw. Aufgabe zu lösen ist.

Bei deklarativen Programmiersprachen wird vom WIE weitgehend abstrahiert und stattdessen eine Beschreibung des gewünschten Ergebnisses : „WAS“ durchgeführt. Das genaue Vorgehen bei der Berechnung des Ergebnisses ist nicht explizit in dieser Beschreibung enthalten. Z.B. sind Spezifikationen in verschiedener Form eher deklarativ. Die logischen und die funktionalen Programmiersprachen rechnet man zu den deklarativen Programmiersprachen.

#### von-Neumann Programmieridee:

**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 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.

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

Wesentliche Elemente sind der Funktionsaufruf und das rekursive Programmieren.

Wichtig:! in nicht-strikten FPS gibt (i.a.) keinen expliziten globalen Zustand, der von den Funktionen verändert werden kann. Somit gibt es keine Zuweisungen.

Zuweisungen, Seiteneffekte und IOs erfordern spezielle Vorkehrungen: In Haskell ist dies durch das monadische Programmierkonzept realisiert.

### Pures Funktionales Programmieren

Prinzip der *referentiellen Transparenz*:

Gleiche Funktion angewendet auf gleiche Argumente ergibt gleichen Wert, d.h. nur die Argumente bestimmen den Wert. Funktionen verändern sich nicht durch die Verarbeitung.

**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 dabei nur der **Wert des Ausdrucks**, nicht die Reihenfolge der Auswertung

#### 1.2.1 Warum behandeln wir Haskell in einer Vorlesung?

Einige Gründe sollen hier aufgezählt werden, ebenso werden wir auch die Nachteile erwähnen.

- Nicht-strikte FPS erlauben die saubere Behandlung von Funktionen höherer Ordnung: Funktionen als Argumente, und als manipulierbare Objekte.
- Die Semantik, operational und denotational, erlaubt sehr leicht zahlreiche korrekt Programmtransformationen ohne hässliche Nebenbedingungen wie: wenn keine Seiteneffekte auftreten, wenn der Aufruf terminiert usw. Das bedeutet, dass Programmtransformationen, Optimierungen einen einfachen Begriff von Korrektheit haben, der auch praktisch dazu führt, dass man Programme in korrekter Weise analysieren, optimieren und verändern kann, ohne Fehler einzuführen.
- Parallelisierbarkeit ist durch einfache Analysen zu erkennen.
- Das Typsystem, parametrisch polymorph und die Erweiterung zum Typklassensystem ist sehr gut durchdacht und sehr weitgehend, so dass der Programmierstil und das Typsystem dazu führt, dass zahlreiche Fehler bereits in der Programmierphase vermieden werden.

- Die aktive Forschergemeinde ist ziemlich groß, so dass in diesem Bereich noch wegweisende Entwicklungen zu erwarten sind.

Nachteile:

- Effizienz zur Laufzeit erfordert hohen Aufwand beim Schreiben eines Compilers.
- Die Schnittstellen zu GUIs, Betriebssystem und anderen IO-Medien sind unhandlich, da hier zwei verschiedene Prinzipien nicht ganz passen: Seiteneffekte, Sequentialität auf der einen und Seiteneffektfreiheit und deklarative, nichtsequentielle Spezifikation. Dies wird teilweise durch das monadische Programmierkonzept zum IO wieder wettgemacht.

### 1.2.2 Kommentar zum Verhältnis FPS zu OO

Funktionale Programmiersprachen haben folgende Relation zu einigen unter dem Stichwort OO versammelten Programmierprinzipien:

**Modularisierung und information-hiding** ist in funktionalen Programmiersprachen als Konzept integriert.

**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

**von-Neumann-Konzepte** Die folgenden von-Neumann-Anteile fehlen in reinen funktionalen Sprachen: Objekt-Identität, (d.h. Objekte, die man auf Pointer-Gleichheit prüfen kann), und Veränderbarkeit von Objekten.

Die objektorientierte Programmiersprache Java hat u.a. folgende Kombination von Eigenschaften:

- Klassen und Methoden
- Ausdrücke mit Wert
- Zuweisungen
- automatische Speicherverwaltung
- Typsystem, aber dynamische Typfehler sind möglich
- keine Pointer-Arithmetik

### 1.2.3 Funktions/Prozedur-aufrufe in anderen Programmiersprachen:

**Definition 1.2.1 Seiteneffekt:** *Änderung des Speichers (der Programmierumgebung) während eines Funktions/ Prozeduraufrufs, wobei diese Änderung nach dem Aufruf noch Auswirkungen hat.*

#### Prozeduraufruf: Methoden bei imperativen Programmiersprachen

$$p(a_1, \dots, a_n)$$

##### Methode 1) (call by name)

Funktion  $p$  bekommt  $n$  Argumente (als Pointer bzw. Speicherplätze) der Rückgabe-Wert ist (bzw. die Rückgabewerte) an einer bekannten Adresse.  
(Seiteneffekte sind notwendig)

##### Methode 2) (call by value)

Funktion  $p$  sieht nur die Argumentwerte, Speicherzugriffe sind 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

pures Prolog, pures Scheme, pures Lisp, pures ML pur = seiteneffektfrei

#### Prozeduraufruf: Methoden bei funktionalen Programmiersprachen

##### Methode 1. (call by name) bzw. nicht-strikt

Funktion bekommt  $n$  Argumente (i.a. als Ausdrücke), und setzt diese in den Rumpf der Funktionsdefinition ein.  
Wert wie bei call-by-need (lazy), aber Ausdrücke werden manchmal mehrfach ausgewertet. Unendliche Listen sind möglich.

##### Methode 2) (call by value) bzw. strikt

Funktion erhält nur die Argumentwerte, und setzt diese in den Rumpf ein.

Keine Mehrfachauswertung, aber auch: keine direkt verfügbaren unendlichen Listen ( Ströme). ML, pures Scheme, pures Lisp, ...

### **Methode 3) (call by need) bzw. verzögert (lazy)**

Funktion sieht die Argumentausdrücke, benutzt aber Sharing beim Auswerten.

Keine Mehrfachauswertung von Ausdrücken, unendliche Listen sind möglich.

Haskell, Clean, Miranda

## **1.2.4 Seiteneffekte**

Wenn Seiteneffekte erlaubt sind, d.h. wenn man in einem Funktions- bzw. Prozeduraufruf Zuweisungen machen darf, dann hat man folgende Eigenschaften, die man je nach Sicht als positive bzw negativ bewerten kann:

- Seiteneffekte werden unterstützt von Hardware, sind effizient; die Übergabe von Werten ist einfach.
- Ein Nachteil ist, dass die Sequentialisierung des Programms durch den Programmierer erforderlich ist, was eine Parallelisierung verhindern bzw. erheblich erschweren kann.

Weitere nachteilige Eigenschaften eines Programms bei Verwendung von Seiteneffekten sind:

- Die Semantik einer Funktion ist nur erklärbar durch Veränderung des globalen Zustandes (i.a. des Hauptspeichers).
- 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

Bei konkurrenten Programmiermodellen, auch wenn diese funktional sind, ist ein gewisses Maß an Effekten, die nicht mit dem rein funktionalen Programmieren verträglich sind, unausweichlich.

Hier gibt es jedoch Kombinationen von Lambdakalkülen und konkurrenter Verarbeitung, oder Kombinationen mit Ein/Ausgabe. Die Theorie dazu kann ähnlich zu reinen funktionalen Programmen aufgebaut werden.

### 1.2.5 Semantik

Programmiersprachen sollten vernünftigerweise eine Semantik haben. D.h. eine formale Definition der Bedeutung der Befehle / Konstrukte. Die Semantik sollte am besten keine offenen Stellen haben (... ist dem Implementierer überlassen..), damit man daraus Schlüsse über das Verhalten (die Semantik) von größeren Programmfragmenten ziehen kann. Natürlich sollte die Semantik auch handhabbar sein.

Mittels einer Semantik kann man die Korrektheit von Programmtransformationen, insbesondere von Optimierungen durch Abänderung des Programms begründen bzw. beweisen.

Es gibt verschiedene Methoden zur Angabe einer Semantik.

**Operationale Semantik** wird verwendet, um den Effekt von Einzelbefehlen auf die (Variablen-) umgebung zu definieren. Ein Interpreter ist meist eine direkte Implementierung der operationalen Semantik. Aber auch eine abstrakte Maschine kann man oft direkt in Relation zu einer operationalen Semantik setzen.

Diese passt auf natürliche Weise zu imperativen Programmiersprachen.

Wir werden in dieser Vorlesung eine operationale Semantik von funktionalen Programmiersprachen definieren und damit eine Gleichheit auf den Programmen durch sogenannte kontextuelle Gleichheit definieren: Zwei Programmfragmente sind gleich, wenn in jedem Programm das eine durch das andere ersetzt werden kann, ohne dass sich die Semantik des Programms ändert. Man verwendet hier oft auch die Schtwaise: ohne dass sich das beobachtbare Verhalten des Programms verändert.

Im Skript wird immer wieder Wert auf Korrektheitsaspekte von Programmtransformationen, Compiler-Schemata und Auswertung von Programmen gelegt. Die Zugang über kontextuelle Gleichheit erlaubt eine einfache Begründung von Regeln und Eigenschaften von Funktionen, insbesondere auch von rekursiven Funktionen.

**Denotationale Semantik** Bildet Anweisungen bzw. Funktionen ab auf Ausdrücke des Lambda-Kalküls bzw eines erweiterten Lambda-Kalküls.

Dies passt gut zu nicht-strikten FPS, aber man kann es auch für imperative Programmiersprachen definieren :

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

Die formale Handhabung einer denotationalen Semantik ist mathematisch schwierig und würde den Rahmen dieser Vorlesung sprengen.

## 1.2.6 Eine Liste einiger funktionaler Programmiersprachen / Programmiersprachen mit funktionalen Möglichkeiten

Wir betrachten folgende 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

Für einige Programmiersprachen sind auch Web-Adressen angegeben. Die Auflistung ist eine Auswahl und nicht vollständig.

**Haskell** nicht-strikt, polymorph statisch stark getypt + Typklassen, patterns, Funktionen höherer Ordnung; keine Seiteneffekte, [www.haskell.org](http://www.haskell.org)

**CLEAN:** nicht-strikt, polymorph statisch stark getypt; Funktionen höherer Ordnung; patterns, keine Seiteneffekte aktive Entwicklung  
[clean.cs.ru.nl](http://clean.cs.ru.nl)

**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, Patterns in manchen Varianten, aktive Entwicklungen: [www.schemers.org](http://www.schemers.org)

**ML:** Standard ML strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; Seiteneffekte, patterns, aktive Entwicklung  
<http://www.smlnj.org/>  
 Variante CAML: [caml.inria.fr/](http://caml.inria.fr/)

**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, Seiteneffekte?

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

**OPAL** strikt, pur, funktional, higher-order, polymorphe Typen, (wie ML), aber zusätzlich algebraische Spezifikationen

**CAL** nicht-strikt, funktional, higher-order, polymorphe Typen, eingebettete Java-Typen, eingeschränkte Seiteneffekte,  
[http://homepage.mac.com/luke\\_e/FileSharing13.html](http://homepage.mac.com/luke_e/FileSharing13.html)

**Miranda** nicht-strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; patterns, keine Seiteneffekte, trademarked by D. Turner.  
<http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/Announce.htm>

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

### 1.2.7 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