Einleitung und Motivation

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 22. Juli 2020 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Klassifizierung von Programmier-paradigmen und -sprachen

Was unterscheidet Funktionale Programmiersprachen von anderen?

Klassifizierung von Programmiersprachen (1)

Imperative Programmiersprachen:

- lat. imperare = befehlen
- Programm = Folge von Anweisungen (Befehlen)
- Ausführung = Sequentielle Abarbeitung der Befehle,
  Befehle ändern den Zustand des Speichers (Wirkung nur über Seiteneffekte)

```
X:=10;
while X < 100 do
     X:=X+1;
```

- Passend zur von-Neumann Rechnerarchitektur
- Programm beschreibt WIE der Zustand modifiziert werden soll
- Prozedurale Sprachen: Strukturierung des imperativen Codes durch Prozeduren

Klassifizierung von Programmiersprachen (2)

Objektorientierte Programmiersprachen:

- Programm: Definition von Klassen
  (Klassen = Methoden + Attribute),
  Strukturierung durch Vererbung
- Ausführung=veränderliche Objekte (Instanzen von Klassen)
  versenden Nachrichten untereinander
  sequentielle Abarbeitung innerhalb der Methoden

```
Fahrzeug auto = new Fahrzeug();
auto.setSpeed(180);
```

- Zustandsänderung des Speichers durch Veränderung der Objekte
- werden oft zu den imperativen Sprachen gezählt

Klassifizierung von Programmiersprachen (3)

Deklarative Programmiersprachen

- lat. declarare = erklären
- Programm = Beschreibt nicht WIE, sondern eher WAS berechnet werden soll.
- Ausführung= I.A. keine Manipulation des Zustands, sondern Wertberechnung von Ausdrücken
- Lassen sich grob in funktionale Programmiersprachen und logische Programmiersprachen aufteilen.

Klassifizierung von Programmiersprachen (4)

Logische Programmiersprachen

- Prominentes Beispiel: Prolog
- Programm = Menge von logischen Formeln
  (in Prolog: Hornklauseln der Prädikatenlogik)

```
bruder(X,Y):- istSohn(X,Z), istSohn(Y,Z), X/=Y.
```

- Ausführung = Herleitung neuer Fakten mittels logischer Schlüsse (z.B. Resolution)
- Ausgabe von (mehreren) Antworten auf Anfragen.

Klassifizierung von Programmiersprachen (5)

Funktionale Programmiersprachen

- Programm = Funktionsdefinitionen + Ausdruck (`main`)

```
quadrat x = x * x
main = quadrat 5
```

- Ausführung = Auswerten des Ausdrucks (`main`).
- Resultat der Auswertung: Ein einziger Wert

Klassifizierung von Programmiersprachen (5)

Funktionale Programmiersprachen (Forts.)

- In reinen funktionalen Programmiersprachen:
  Keine (sichtbare) Manipulation des Zustands
- D.h. keine Seiteneffekte, es gilt referentielle Transparenz
- Variablen stellen keine Speicherplätze dar, sondern unveränderliche Werte

Referentielle Transparenz

Gleiche Funktion auf gleiche Argumente liefert stets das gleiche Resultat

Vorteile funktionaler Programmierung

Welche Motivation gibt es, sich diese Sprachen anzuschauen?
Welche Vorteile bieten diese Sprachen?

Warum Funktionale Programmierung? (1)

- **Andere Sichtweise der Problemlösung:**
  abstraktere Lösungsformulierung, auch hilfreich beim imperativen Programmieren
- **Elegantere Problemlösung:**
  Funktionale Programme sind meistens kürzer und verständlicher, da „mathematischer“
- **Mathematisch einfacher handhabbar:**
  Aussagen wie “mein Programm funktioniert korrekt” oder “Programm A und Programm B verhalten sich gleich” relativ einfach nachweisbar
- **Weniger Laufzeitfehler bei starker und statischer Typisierung:**
  Fehlerentdeckung zur Compilezeit

Warum Funktionale Programmierung? (2)

- **Keine Fehler durch Speicherüberschreiben**
- **Debuggen einfach:** Fehlerreproduktion unabhängig vom Speicherzustand möglich
- **Testen einfach:** Unit-Tests einfach: Funktionen können unabhängig getestet werden
- **Wiederverwendbarkeit:** Funktionen höherer Ordnung haben hohen Abstraktionsgrad.
- **Compiler Optimierungen:** als korrekt nachweisbar via Semantik.

Funktion höherer Ordnung

Eingabeparameter (und Ausgabe) dürfen selbst Funktionen sein

Warum Funktionale Programmierung? (3)

Beispiel: Summe und Produkt der Elemente einer Liste

Warum Funktionale Programmierung? (3)

Beispiel: Summe und Produkt der Elemente einer Liste

```
summe liste =
  if leer(liste) then 0 else
    „erstes Element der Liste“ + (summe „Liste ohne erstes Element“)
```

Warum Funktionale Programmierung? (3)

Beispiel: Summe und Produkt der Elemente einer Liste

`summe` liste =
  if leer(liste) **then** 0 **else**
    „erstes Element der Liste“ + (`summe` *„Liste ohne erstes Element“*)

`produkt` liste =
  if leer(liste) **then** 1 **else**
    „erstes Element der Liste“ * (`produkt` *„Liste ohne erstes Element“*)

Warum Funktionale Programmierung? (3)

Beispiel: Summe und Produkt der Elemente einer Liste

```
summe liste =
  if leer(liste) then 0 else
    „erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

produkt liste =
  if leer(liste) then 1 else
    „erstes Element der Liste“ * (produkt „Liste ohne erstes Element“)
```

Warum Funktionale Programmierung? (3)

Beispiel: Summe und Produkt der Elemente einer Liste

```
summe liste =
  if leer(liste) then 0 else
     „erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

produkt liste =
  if leer(liste) then 1 else
     „erstes Element der Liste“ * (produkt „Liste ohne erstes Element“)
```

Besser abstrahiert:

```
reduce e op liste =
  if leer(liste) then e else
     „erstes Element der Liste“ op  (reduce e op „Liste ohne erstes Element“)
```

summe und produkt sind dann nur Spezialfälle:

```
summe liste   = reduce 0 (+) liste
produkt liste = reduce 1 (*) liste
```

Warum Funktionale Programmierung? (4)

- **Einfache Syntax:** I.A. haben funktionale Programmiersprachen wenige syntaktische Konstrukte / Schlüsselwörter
- **Modularität:** Die meisten FP bieten Modulsysteme zur Strukturierung und Kapselung des Codes.
- **Kompositionalität:** Funktionskomposition erlaubt strukturiertes Zusammensetzen großer Funktionalitäten
- **Parallelisierbarkeit:** Reihenfolge der Auswertung von funkt. Programmen oft irrelevant, bis auf Datenabhängigkeit, daher gut parallelisierbar.

Warum Funktionale Programmierung? (5)

- **Multi-Core Architekturen**: FP "im Trend":
  Seiteneffektfreiheit impliziert:
  Race Conditions oder Deadlocks können (im puren Teil) nicht auftreten, da diese direkt mit Speicherzugriffen zusammenhängen.
- **Unveränderliche Datenstrukturen** (Konsequenz der referentiellen Transparenz)
  Beispiel: Einfügen in eine Liste: Erzeuge neue Liste
  - Gemeinsame Teile der alten und neuen Liste nur einmal speichern
  - Automatische Garbage Collection entfernt alte Versionen
  - Typische Fallen von veränderlichen Strukturen treten nicht auf
    (z.B. Veränderung des Iterators während einer for-Iteration in Java)
  - Beim Rekursionsrücksprung existieren die alten Zustände noch
  - Nebenläufigkeit: explizites Kopieren von Datenstrukturen zwischen Threads nicht notwendig

Warum Funktionale Programmierung? (6)

- **Starke Typisierung**: Aufdeckung von Fehlern, Auflösung von Überladung, Dokumentation von Funktionen, Suche passender Funktionen. Typen können oft automatisch inferiert werden. Fortgeschrittene Typsysteme dienen der Verifikation (z.B. in Agda).
- **Große Forschergemeinde, neueste Entwicklungen**
- Auch moderne imperative Programmiersprachen übernehmen **FP-Konzepte**, z.B. Java Generics, etc.
- **Concurrent Haskell** verbindet pures funktionales Programmierung mit Nebenläufigkeit.
- **Software Transactional Memory:** Starke Typisierung hilft bei Korrektheit.

Eigenschaften von funktionalen Programmiersprachen

Anhand welcher Eigenschaften lassen sich FP unterscheiden?

Wesentliche Eigenschaften von FP (1)

Pure vs. Impure:

- Pure FP: Keine Seiteneffekte erlaubt
- Impure FP: Seiteneffekte möglich
- Daumenregel:
  Pure FP = nicht-strikte Auswertung
  Impure FP = strikte Auswertung

- Haskell: pure FP + (sauber abgetrennte Seiteneffekte)
  nicht-strike Auswertung

Wesentliche Eigenschaften von FP (2)

strikte vs. nicht-strikte Auswertung

- Strikte Auswertung (auch call-by-value oder applikative Reihenfolge):
  Definitionseinsetzung nach Argumentauswertung
- Nicht-strikte Auswertung (auch call-by-name, call-by-need, lazy, verzögert, Normalordnung):
  Definitionseinsetzung ohne Argumentauswertung.

Wesentliche Eigenschaften von FP (2)

strikte vs. nicht-strikte Auswertung

- Strikte Auswertung (auch call-by-value oder applikative Reihenfolge):
  Definitionseinsetzung nach Argumentauswertung
- Nicht-strikte Auswertung (auch call-by-name, call-by-need, lazy, verzögert, Normalordnung):
  Definitionseinsetzung ohne Argumentauswertung.

Beispiel: `quadrat` $x = x * x$

- Strikte Auswertung von `quadrat` $(1 + 2)$:
  `quadrat` $(1 + 2) \rightarrow$ `quadrat` $3 \rightarrow 3 * 3 \rightarrow 9$
- Nicht-Strikte Auswertung von `quadrat` $(1 + 2)$
  `quadrat` $(1 + 2) \rightarrow (1 + 2) * (1 + 2) \rightarrow 3 * (1 + 2) \rightarrow 3 * 3 \rightarrow 9$

Wesentliche Eigenschaften von FP (3)

Eigenschaften des Typsystems

stark vs. schwach:

- stark: Alle Ausdrücke müssen getypt sein
- schwach: (Manche) Typfehler werden vom Compiler akzeptiert bzw. manche Ausdrücke dürfen ungetypt sein

dynamisch vs. statisch:

- statisch: Typ des Programms muss zur Compilezeit feststehen.
  Konsequenzen: Keine Typfehler zur Laufzeit, Typinformation zur Laufzeit unnötig
- dynamisch: Typermittlung zur Laufzeit
  Konsequenzen: Typfehler zur Laufzeit möglich, Typinformation zur Laufzeit nötig.

monomorph vs. polymorph:

- monomorph:: Funktionen / Ausdrücke haben festen Typ
- polymorph: Schematische Typen (mit Typvariablen) erlaubt.
  Funktionen haben viele Typen

Wesentliche Eigenschaften von FP (4)

First-Order vs. Higher-Order Funktionen

- first-order: Argumente von Funktionen müssen Datenobjekte sein
- higher-order: Auch Funktionen als Argumente erlaubt

Speicherverwaltung

- automatische Speicherbereinigung (Garbage Collector): Benutzter Speicher wird automatisch freigegeben.

Überblick über verschiedene funktionale Programmiersprachen

Eine (unvollständige) Aufzählung von verschiedenen funktionalen Programmiersprachen

Überblick Funktionale Programmiersprachen (1)

- **Lisp**: Sprachfamilie, List processing, Ende 1950er Jahre, von John McCarthy). Sprache in Anlehnung an den Lambda-Kalkül, ungetypt, strikt.
- **Common Lisp**: Lisp-Dialekt, erste Ausgabe 1984, endgültiger Standard 1994, dynamisch typisiert, auch OO- und prozedurale Anteile, Seiteneffekte erlaubt, strikt,
- **Scheme**: Lisp-Dialekt, streng und dynamisch getypt, call-by-value, Seiteneffekte erlaubt, eingeführt im Zeitraum 1975-1980, letzter Standard aus dem Jahr 2007,
- **Clojure**: relativ neue Sprache, Lisp-Dialekt, der direkt in Java-Bytecode compiliert wird, dynamisch getypt, spezielle Konstrukte für multi-threaded Programmierung (software transactional memory system, reactive agent system)

Überblick Funktionale Programmiersprachen (2)

- **ML**: Sprachfamilie, Meta Language, statische Typisierung, Polymorphismus, automatische Speicherbereinigung, i.A. call-by-value Auswertung, Seiteneffekte erlaubt, entwickelt von R. Milner 1973
- **SML**: Standard ML, ML-Variante: call-by-value, entwickelt 1990, letzte Standardisierung 1997, Sprache ist vollständig formal definiert, Referenz-Implementierung: Standard ML of New Jersey
- **OCaml**: Objective Caml, Weiterentwicklung von Caml (Categorically Abstract Machine Language), ML-Dialekt, call-by-value, stark und statische Typisierung, polymorphes Typsystem, automatische Speicherbereinigung, nicht Seiteneffekt-frei, unterstützt auch Objektorientierung, entwickelt: 1996
- **F#**: Funktionale Programmiersprache entwickelt von Microsoft ab 2002, sehr angelehnt an OCaml, streng typisiert, auch objektorientierte und imperative Konstrukte, call-by-value, Seiteneffekte möglich, im Visual Studio 2010 offiziell integriert

Überblick Funktionale Programmiersprachen (3)

- **Haskell**: Pure funktionale Programmiersprache, keine Seiteneffekte, strenges und statisches Typsystem, call-by-need Auswertung, Polymorphismus, Komitee-Sprache, erster Standard 1990, Haskell 98: 1999 und nochmal 2003 leichte Revision, neuer Standard Haskell 2010 (veröffentlicht Juli 2010),
- **Clean**: Nicht-strikte funktionale Programmiersprache, stark und statisch getypt, polymorphe Typen, higher-order, pure

Überblick Funktionale Programmiersprachen (4)

- **Erlang**: Strikte funktionale Programmiersprache, dynamisch typisiert, entwickelt von Ericsson für Telefonnetze, sehr gute Unterstützung zur parallelen und verteilten Programmierung, Ressourcen schonende Prozesse, entwickelt ab 1986, open source 1998, hot swapping (Code Austausch zur Laufzeit), Zuverlässigkeit "nine nines"= 99,9999999 %
- **Scala**: Multiparadigmen Sprache: funktional, objektorientiert, imperativ, 2003 entwickelt, läuft auf der Java VM, funktionaler Anteil: Funktionen höherer Ordnung, Anonyme Funktionen, Currying, call-by-value Auswertung, statische Typisierung
- **Agda:** funktional, dependent types, automatisches Beweissystem zur Verifikation von Programmen, Haskell-ähnliche Syntax, aber eine andere Semantik. Die Sprache ist total, d.h. Funktionen müssen terminieren.