Einführung in die Funktionale Programmierung

Am Beispiel von OCaml

Michael Dienert

12. Oktober 2018

Inhaltsverzeichnis

1 Welche funktionale Sprache wählen wir?

1.1 Welche funktionale Sprache soll ich lernen?

Eine kleine Auswahl an funktionalen Sprachen:

Lisp : Lisp ist die zweitälteste Hochsprache (1958), die heute noch verwendet wird und Bedeutung hat (nach FORTRAN)

Scheme : Scheme ist ein Lisp-Dialekt und wird vor allem am MIT (*Massachusetts Institute of Technology*) in der Lehre benutzt. Das zugehörige Lehrbuch (*Structure and Interpretation of Computer Programs*) ist frei verfügbar, somit eignet sich Scheme sehr gut um funktionales Programmieren zu erlernen.

Haskell: Haskell ist eine strikt funktionale Sprache, was sie für den Einsteiger weniger geeignet macht. Z.B. sind IO-Operationen immer mit einer Zuweisung verbunden und damit rein funktional nicht darstellbar.

Weitere funktionale Sprachen:

OCaml: OCaml hat neben funktionalen auch imperative Eigenschaften. OCaml wird vom französchischen *Institut national de recherche en informatique et en automatique (INRIA)* entwickelt.

F#: F# ist stark von OCaml beeinflusst. Man soll teilweise sogar OCaml-Code mit dem F#-Compiler übersetzen können. Da die Entwicklung von F# aber von *Mordor* aus gesteuert wird, scheidet es für uns aus und wir wählen OCaml.

Zugegeben, die Bedeutung von OCaml geht nach einem Hype so um ca. 2005 herum zurück. Aber hier geht es um den Einstieg in die funktionale Programmierung, Scheme darf ich nicht nehmen, und dann beleibt immer noch die F#-Option, für die, die ihre Seele verkaufen wollen.

2 Einführung: wo liegt der Unterschied zur imperativen Programmierung

2.1 Imperative Programmierung

- Imperative Programmierung orientiert sich an der *Funktionsweise der CPU* $\Rightarrow$ Speichermodell, Rechenwerk, Ein- / Ausgabe. Beispiele: C, C++, Java, Pascal, Fortran, Algol, ...

Variablen sind nichts anderes als gut lesbare Namen für *Speicheradressen*

Kontrollstrukturen bringen die `jmp`-Befehle des Prozessors in eine für Menschen lesbare Form.

Funktionen oder *Methoden* sind nichts anderes als *Unterprogramme*, zu denen das Hauptprogramm mit `jmp`-Befehlen verzweigt.

- Algorithmen werden mit Kontrollstrukturen und Anweisungen (z.B. i=i+1;) formuliert
- Die Anweisungen können globale und lokale Variablen und damit den *Maschinenzustand ändern*.
- **Problem**: der Maschinenzustand hängt von der *Reihenfolge der Ausführung* und den *Ausgangswerten der Variablen* ab
- Bei komplexen Algorithmen ist es so gut wie unmöglich vorherzusagen, ob ein ungewollter Maschinenzustand auftreten kann.

2.2 Funktionale Programmierung

- Die funktionale Programmierung orientiert sich am *mathematischen Funktionsbegriff*

  Funktion: eine Funktion ordnet jedem Element der Definitionsmenge **D** genau ein Element der Ergebnismenge **Z** zu.

- Die Algorithmen einer funktionalen Programmiersprache basieren auf *Ausdrücken*, die ausgewertet werden.
- Ein Ausdruck ist eine Kombination von *Funktionen*, *Operatoren* und *Werten*, die einen *Wert* zurückliefern, also *ausgewertet* werden können. Beispiele:
  - `16 + 3 * 8`
  - `sqrt 2.0`
  - `2 < 3`
  - `2.0 > 3.0`
  - `2 + sum 3 4`
- Die Variablen einer funktionalen Programmiersprache bezeichnen *Werte* und keine Speicheradressen! $\Rightarrow$ **Es gibt keine Zuweisung!!!**
- Es erfolgt kein *Ablauf von Anweisungen*, sondern eine *Auswertung von Ausdrücken*.
- Die Auswertung ist eine mathematische *Gleichungsumforumung*

2.2.1 Einige OCaml-Operatoren

```
let dreifach (x:float) = 3.0 *. x
let inc (x:float) = x +. 1.0
Anwendung:
inc(dreifach 7.0 -. inc 2.0)
Umformung:
inc(3.0 *. 7.0 -. 2.0 +. 1.0)
inc(21.0 -. 3.0)
inc(18.0)
(18.0 +. 1.0)
19
```

- Kurzer erster OCaml-Syntax-Einschub: es gibt keine überladenen Operatoren!
- es gibt Operatoren fuer Rechnen mit Integerwerten: + , - , * , /
- und entsprechend Operatoren fuer das Rechnen mit Fliesskommazahlen: +. , -. , *. , /.
- Schleifen in imperativen Programmiersprachen arbeiten mit Zuweisungen
- Da es in der funktionalen Programmierung keine Zuweisungen gibt, gibt es auch keine Schleifen
- Wiederkehrende Operationen können in *rein funktionalen* Sprachen nur mittels *Rekursion* gelöst werden.

3 Recursion

3.1 Rekursionsbeispiel

```
let rec fakultaet (n:int) =
if n=0 || n=1 then 1
else n * fakultaet (n-1);;
```

```
Auswertung mit Gleichungsauflösung
fakultaet 3
= 3 * ( fakultaet(2) )
= 3 * ( 2 * ( fakultaet(1) ) )
= 3 * ( 2 * (1) )
= 3 * ( 2 * 1 )
= 3 * ( 2 )
= 3 * 2
= 6
```

Nachteil: Speicherbedarf wächst linear mit der Anzahl an Rekursionsschritten

3.2 Endrekursion

Das Falkultätsbeispiel mit *Endrekursion*. Hier ist der Speicherbedarf unabhängig von der Anzahl der Rekursionen:

```
let facultaet n =
  let rec facHelp n akku =
     if n = 0 then akku
     else facHelp (n-1) (n*akku)
  in facHelp n 1;;
```

```
facultaet 3
= facHelp 3 1
= facHelp (3-1) (3*1)
= facHelp  2     3
= facHelp (2-1) (2*3)
= facHelp  1     6
= facHelp (1-1) (1*6)
= facHelp  0     6
= 6
```

4 Wichtige Eigenschaften von OCaml

4.1 Funktionsaufrufe, Typisierung

- Funktionsaufrufe finden **ohne Klammern** und **ohne Kommas** statt! Klammern sind nur notwendig, damit der Compiler Ausdrücke in der richtigen Reihenfolge auswerten kann. Beispiel:

```
f 5 (g "hello") 3     (* f hat drei argumente, g hat ein
    argument *)
f (g 3 4)             (* f hat ein argument, g hat zwei
    argumente *)
```

- Typdeklarationen: *bezeichner : typ*
- OCaml ist *streng* typisiert
- OCaml arbeitet mit *impliziter* Typisierung:

```
# let f a b = a +. b;;
val f : float -> float -> float = <fun>
```

4.2 let-Bindings

- Deklarationen müssen das Schlüsselwort `let` verwenden:

```
let name = expression
```

- mit `let` wird ein beliebiger *Ausdruck* an einen Bezeichner **gebunden**. $\Rightarrow$ let-Binding
- Der Bezeichner wird somit zum *Aliasnamen* für den gesamten Ausdruck. Überall wo im Folgenden der Bezeichner steht, kann man sich den gesamten Ausdruck eingesetzt denken.

- Bindungen *innerhalb* eines Ausdrucks, also lokale Bindungen werden mit dem Schlüsselwort **`in`** deklariert:

```
let name = expression in ... ;;
```

  Hier gilt die Bindung nur ab dem Schlüsselwort **`in`** bis zu den doppelten Strichpunkten **`;;`**

- Eine Funktion kann erst aufgerufen werden, wenn sie zuvor komplett deklariert wurde. Das geht bei rekursiven Aufrufen natürlich nicht, weshalb hier mit **`let rec`** gearbeitet werden muss.

Nochmals das Falkultätsbeispiel mit *Endrekursion*, diesmal aber *ohne* lokale Deklaration:

```
(*diesmal wird die hilfsfunktion global deklariert *)
let rec facHelp n akku =
  if n = 0 then akku
  else facHelp (n-1) (n*akku);;

let facultaet n =
  facHelp n 1;;

facultaet 5;;
```

4.3 Generische Typisierung

OCaml erlaubt auch Funktionen mit generischem Rückgabetyp.

Beispielcode:

```
let x_oder_y testfunktion x y =
  if testfunktion x then x else y;;
val x_oder_a : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

Der Compiler kann hier noch nicht festlegen:

- Welchen Typ das Argument der `testfunktion` hat
- Welchen Typ die Argumente von `x_oder_y` haben
- Welchen Typ der Rückgabewert hat
- OCaml verwendet daher eine sog. *type variable*: `'a`

Zwei Testfunktionen:

```
let groesserDrei x =
  x > 3;;

let laengerDrei s =
  String.length s > 3;;
```

und das Ergebnis der Aufrufe:

```
x_oder_y groesserDrei 1 (-7);;
- : int = -7

x_oder_y laengerDrei "fooooooo" "bar";;
- : string = "fooooooo"
```

4.4 Datenstrukturen

4.4.1 Tupel, Records, Listen und Pattern Matching

- OCaml stellt grundsätzlich zwei Datenstrukturen bereit: Tupel und Listen
- Records sind eine Sonderform der Tupel. Dazu später mehr.
- Tupel sind Sammlungen von Werten *verschiedenen* Typs.
- Die Anzahl der Elemente eines Tupels ist *fest*.
- Listen sind Sammlungen von Elementen *gleichen* Typs.
- Die Anzahl der Elemente einer Liste ist beliebig.
- Auf die Elemente eines Tupels oder einer Liste kann mit *pattern matching* zugegriffen werden.
- Pattern Matching ist eine erweiterte Form der *switch-case* Anweisung aus C und Java.
- alles Weitere folgt in Code-Beispielen

4.5 Beispiele zu Tupel, Listen, Pattern-Matching

4.5.1 Beispiele zu Tupeln

- Die einzelnen Werte eines Tupels werden duch Kommas getrennt:

```
let a_tuple = (3,"three");;
```

- Mit Patter-Matching werden die Einzelwerte ermittelt:

```
let (x,y) = a_tuple;;
```

- Das Pattern hat hier den Wert `(x,y)`
- Welchen Wert und welchen Typ haben x und y?

4.5.2 Beispiele zu Pattern-Matching

- Noch ein Beispiel mit Pattern-Matching:

```
let distance (x1,y1) (x2,y2) =
sqrt ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.);;
let p1 = (1.0,0.0);;
let p2 = (0.0,1.0);;
distance p1 p2;;
```

- Welcher Wert wird angezeigt?

4.5.3 Beispiele zu Listen

- Die einzelnen Werte einer Liste werden durch Strichpunkte (Semikolon) getrennt:

```
let liste = [1;2;3];;
(*alternativ*)
let liste = 1::2::3::[];;
```

- der Operator `::` hängt ein Element **vor** die Liste:

```
let listeNeu = 0::liste;;
```

- der Operator `@` verkettet zwei Listen:

```
let listeLang = [1;2;3] @ [4;5;6]
liste @ listeNeu
```

4.5.4 Beispiele zu Pattern-Matching und Listen

- Pattern-Matching mit dem Schlüsselwort `match` und der Zeichenkombination `->`

```
let meineLieblingsSprache languages =
   match languages with
   | erstesElement :: derGanzeRest -> erstesElement
   | [] -> "OCaml" (* A good default! *)
;;

meineLieblingsSprache ["Alemannisch";"English";"Spanish";"
   French"];;
meineLieblingsSprache [];;
```

4.5.5 Eine Übungsaufgabe zu Pattern-Matching, Listen und rekursiven Funktionen

- Schreibe eine *rekursive* Funktion `sum`, die alle Werte einer Integer-Liste aufsummiert
- Hinweis: Pattern-Matching mit `match` verwenden.
- Schreibe eine Funktion *trimmDoppler*, die zwei benachbarte Elemente, die den gleichen Wert haben, auf ein einzelnes Element reduzieren.

4.5.6 Lösung

```
let rec sum liste =
   match liste with
   | head :: tail -> head + sum tail
   | [] -> 0
;;
```

```
let rec trimDoppler liste =
 match liste with
 |[] -> []
 |[head] -> [head]
 |e1 :: e2 :: rest ->
  if e1 = e2 then trimDoppler (e1::rest)
  else e1 :: trimDoppler (e2::rest)
;;
```

4.6 Quicksort

- Quicksort arbeitet nach dem römischen Prinzip: *Teile und herrsche!*
- Um eine Liste zu sortieren, wird diese in zwei Listen geteilt, die einzeln sortiert werden.
- Das Teilen erfolgt nach diesem Algorithmus:
  - Wähle ein beliebiges Element der Liste, z.B. das *Erste* (head)
  - Dieses Element nennen wir *Pivot-Element*
  - Verschiebe alle Elemente der Liste, die kleiner sind als *Pivot* in die linke Teilliste
  - Verschiebe alle Elemente der Liste, die grösser sind als *Pivot* in die rechte Teilliste.
- Bearbeite die Teillisten nach dem gleichen Prinzip (rekursiv)

4.7 Quicksort: Zahleneispiel

```
      [10;13;17;4;5;2;7;3;1;11]
      [4;5;2;7;3;1]--[10]--[13;17;11]
                          [11]--[13]--[17]
                                  []--[17]--[]
                    []--[11]--[]

      [2;3;1]--[4]--[5;7]
                []--[5]--[7]
                   []--[7]--[]

      [1]--[2]--[3]
            []--[3]--[]
[]--[1]--[]

[1;2;3;4;5;7;10;11;13;17]
```

5 Funktionen im Detail

5.1 Funktionsdeklaration mit `fun`

- Bisher haben wir Funktionen so deklariert:

```
let inc a  = a + 1;;
```

- Das ist eine vereinfachte Syntax (syntactic sugar) für die ausführliche Deklaration:

```
let inc = fun a -> a + 1 ;;
```

- Wir haben gelernt, dass mit `let` Aliasnamen für Ausdrücke gebildet werden.
- Das heisst umgekehrt, dass die Deklaration mit `fun` einen ganz *normalen, gleichberechtigten Ausdruck* darstellt:

```
fun a -> a + 1 ;;
```

Das bedeutet:

- Funktionsausdrücke kann man mit `let` an einen Namen binden: klar, hatten wir schon.
- Funktionsausdrücke können als Parameter anderer Funktionen auftreten.
- Funktionsausdrücke können als Ergebnis einer Funktion zurückgegeben werden.
- Funktionsausdrücke können in Datenstrukturen wie z.B. Listen oder Tupel gespeichert werden.

Beispiel:

```
let funkliste =
 [ (fun x->x+1); (fun x->x+2); (fun x->x+3) ] ;;

let rec iterate l =
   match l with
   |[] -> 0
   |h::t -> print_int (h 5); iterate t;;

iterate funkliste;;
```

5.2 Currying: Funktionen mit mehreren Argumenten

- Funktionen in imperativen Sprachen sind nichts anderes wie Subroutinen
- Deren Parameter müssen beim Funktionsaufruf alle gleichzeitig festliegen

- In der Mathematik und in funktionalen Sprachen gilt das nicht:

  $$f(x) = x \cdot y + z$$

  gleichzeitiges Anwenden der Paramter:

  $$f(3, 4, 5) = 3 \cdot 4 + 5 = 17$$

  partielle Anwendung:

  $$f(3, y, z) = g(y, z) = 3 \cdot y + z$$

  partielle Anwendung auf die neue Funktion g:

  $$g(4, z) = h(z) = 3 \cdot 4 + z$$

5.3 Currying: Funktionen mit mehreren Argumenten

- Die Umwandlung einer Funktion mit mehreren Argumenten in eine Funktion mit nur einem Argument wird *Currying* oder auch *Schönfinkeln* genannnt.
- Namensgeber: Die Mathematiker Haskell Curry (US), Moses Isajewitsch Schönfinkel (RU)

5.4 Currying: Funktionen mit mehreren Argumenten

Ein Beispiel in OCaml:

```
let avg a b = (a+b)/2;;

let avg = (fun a -> (fun b -> (a+.b) /. 2.));;
```

5.5 Ist Java 8 eine funktionale Sprache?

Zwei Artikel (englisch):

- http://java.dzone.com/articles/ whats-wrong-java-8-currying-vs
- http://docs.oracle.com/javase/tutorial/java/ javaOO/lambdaexpressions.html

5.6 Ist Java 8 eine funktionale Sprache?

Lösung zum Oracle-Beispiel in Ocaml:

```
let rec filter list f g =
  match list with
  |[] -> []
  |(mail, laenderCode, geburtsJahr)::t ->
    if ((f laenderCode) && (g geburtsJahr))
    then (mail, laenderCode, geburtsJahr) :: filter t f g
    else filter t f g
;;

filter persoList (fun x -> x="us") (fun x -> x>1950));;
filter persoList (fun x -> x="de") (fun x -> x=1910));;
```