

Objektorientierte Programmiersprachen

Martin Gasbichler

Holger Gast

20.10.2005

Objective Caml

Kennzeichen:

- Mitglied der ML-Familie
- Funktional: Funktionen sind Werte erster Klasse
- Typsystem mit Typinferenz und Polymorphie
- OO-Erweiterungen

Homepage: <http://www.ocaml.org/>

Ocaml Werkzeuge

- `ocaml` Interaktiver Interpreter
- `ocamlc` Byte-Code-Compiler (schnell)
- `ocamlopt` Native-Code-Compiler (sauschnell)

Interaktiver Interpreter

- Prompt: #
- Eingabe beenden mit ; ;
- Antwort: Wert und Typ

Innerhalb von Modulen

Wie im Interpreter, aber

- Kein Prompt
- Keine `;;` am Ende

Basistypen

```
# 1+2*3;;
```

```
- : int = 7
```

```
# (1 + 2)*3;;
```

```
- : int = 9
```

```
# 4.0 *. 3.2 ;;
```

```
- : float = 12.8
```

```
# "abc";;
```

```
- : string = "abc"
```

```
# true;;
```

```
- : bool = true
```

```
# false || true;;
```

```
- : bool = true
```

Aufruf eingebauter Funktionen

Funktionsname, gefolgt von Argumenten

Keine Klammern um die Argumente!

```
# 4.0 *. atan 1.0;;  
- : float = 3.14159265358979312
```

```
# int_of_char 'A';;  
- : int = 65
```

```
# not true;;  
- : bool = false
```

```
# max 12 13;;  
- : int = 13
```

Variablenbindung

```
let var = expr
```

bindet globale Variable *var* an den Wert von Ausdruck *expr*

```
# let pi = 4.0 *. atan 1.0;;
```

```
val pi : float = 3.14159265359
```

```
# let foo = 42;;
```

```
val foo : int = 42
```

Variablen an Funktionen binden

```
let var var1 ... = expr
```

bindet globale Variable *var* an Funktion mit Parametern *var1 ...* und Rumpf *expr*

```
# let squareChar x = int_of_char x * int_of_char x;;  
val squareChar : char -> int = <fun>
```

```
# let doubleAdd x y = (2 * x) + (2 * y);;  
val doubleadd : int -> int -> int = <fun>
```

Funktionsaufrufe

Genau wie bei eingebautern Funktion:

Funktionsname, gefolgt von Argumenten

```
# squareChar 'A';;  
- : int = 4225
```

```
# doubleAdd 3 7;;  
- : int = 20
```

Zusammengesetzte Argumente immer mit Klammern zusammenfassen

```
# doubleAdd (1 + 8) (squareChar 'A');;  
- : int = 8468
```

Funktionstypen

`char -> int`: Funktion, die ein Argument vom Typ `char` nimmt und `int` zurückgibt

```
# let intBoolToString i b =  
    string_of_int i ^ string_of_bool b  
val intBoolToString : int -> bool -> string = <fun>
```

Typ bedeutet: Funktion, die ein Argument vom Typ `int` und ein Argument vom Typ `bool` nimmt und `string` zurückgibt

```
# intBoolToString 23 true;;  
- : string = "23true"
```

Weitere Parameter analog, z.B.. `int -> int -> int -> int`

Funktionsaufrufe, die ganze Wahrheit

Funktionsaufrufe können auch unvollständig sein

```
# intBoolToString 23;;  
- : bool -> string = <fun>
```

Ergebnis ist dann eine Funktion, welche die restlichen Parameter akzeptiert:

```
# let ttBoolToString = intBoolToString 23;;  
val ttBoolToString : bool -> string = <fun>
```

```
# ttBoolToString false;;  
- : string = "23false"
```

Funktionstypen, die ganze Wahrheit

`int -> bool -> string` bedeutet also eigentlich:

Funktion, die ein Argument vom Typ `int` nimmt und eine Funktion vom Typ `bool -> string` zurückgibt

Also:

`int -> bool -> string = int -> (bool -> string)`

Analog:

`int -> int -> int -> int =`
`int -> (int -> (int -> int))`

if

Syntax: `if expr then expr else expr`

`if` gibt Wert zurück:

```
# if 2 < 4 then "okay" else "was is nu los?"  
- : string = "okay"
```

if

Konsequente und Alternative müssen den gleichen Typ haben:

```
# if 2 < 4 then 2 else false;;
```

Characters 21-26:

```
  if 2 < 4 then 2 else false;;  
                ^^^^
```

This expression has type bool
but is here used with type int

Unit

`()` ist einziger Wert mit Typ `unit`

Wird bei Seiteneffekten verwendet

```
# print_string "hello";;  
hello- : unit = ()
```

Sequenzierung

expr1 ; expr2

Wertet erst *expr1* aus und dann *expr2*

```
# print_string "Hi!" ; print_newline ();;
```

```
Hi!
```

```
- : unit = ()
```

```
# 1;2;;
```

```
Characters 0-1:
```

```
Warning: this expression should have type unit.
```

```
  1;2;;
```

```
  ^
```

```
- : int = 2
```

if ohne Alternative

Syntax: `if expr then expr`

`expr` muss Typ `unit` haben

```
# if 2 < 4 then print_string "okay"  
okay- : unit = ()
```

Rekursive Funktionen

`let rec` bindet Variablen rekursiv, d.h. auch in der rechten Seite

```
# let rec fib n =  
    if n < 2  
    then 1  
    else fib(n-1) + fib(n-2);;  
val fib : int -> int = <fun>  
  
# fib 10;;  
- : int = 89
```

Wechselseitig rekursive Funktionen

let rec ... and ... bindet wechselseitig rekursive Funktionen

```
# let rec even x =
  if x=0
  then true
  else odd (x-1)
and odd x =
  if x=0
  then false
  else even (x-1);;
val even : int -> bool = <fun>
val odd  : int -> bool = <fun>

# odd 12;;
- : bool = false
```

Tupel

Zusammengesetzter Typ fester Länge

```
# (1,true);;
```

```
- : int * bool = (1, true)
```

```
# ('A',"bc",4);;
```

```
- : char * string * int = ('A', "bc", 4)
```

Anwendungen:

- Mehrere Werte zurückgeben
- Mehrere Werte zusammenfassen

Polymorphie

Eine Funktion ist polymorph, wenn sie Argumente beliebigen Typs akzeptiert

```
# let const42 x = 42;;  
val const42 : 'a -> int = <fun>
```

```
# let identity x = x;;  
val identity : 'a -> 'a = <fun>
```

Im Typ der Funktion ist `'a` dabei eine *Typvariable*, sie steht für einen beliebigen Typ:

```
# const42 'A';;  
- : int = 42  
# const42 ();;  
- : int = 42
```

```
# identity 1;;  
- : int = 1  
# identity true;;  
- : bool = true
```

Polymorphe Funktionen mit Tupeln

Auch innerhalb von Tupeltypen können Typvariablen auftreten

```
# let pair x y = (x,y);;
```

```
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

```
# let triple x = (x,x,x);;
```

```
val triple : 'a -> 'a * 'a * 'a = <fun>
```

Beobachtung zur Polymorphie

Eine Funktion ist immer dann polymorph in einem Argument, wenn sie nicht in das Argument "hineinschaut"

In einem solchen Fall macht sie keine Voraussetzungen an den Typ, funktioniert also mit allen Typen

Analog fortgesetzt auf Datenstrukturen

Tupeltypen

Länge ist im Typ für Tupel enthalten

$t1 * t2$ und $t1 * t2 * t3$ sind also verschiedene Typen

```
# if true then (1,2) else (1,2,3);;
```

Characters 24-31:

```
    if true then (1,2) else (1,2,3);;
```

```
                ^^^^
```

This expression has type `int * int * int` but
is here used with type `int * int`

Listen

Zentraler Datentyp

Typ: *inhalt list*

z.B.: `int list, bool list`

Anwendung: Beliebige viele Werte gleichen Typs zusammenfassen

Listen erstellen

- Literal: [*expr*;...]
[1;2;3];;
- : int list = [1; 2; 3]
['A';'B'];;
- : char list = ['A'; 'B']
- Konstruktor :: hängt ein Element vorne an die Liste an
1::[];;
- : int list = [1]
'A'::'B'::[];;
- : char list = ['A'; 'B']

Listen, formal

Induktiv definiert:

- Die leere Liste ($[]$) ist eine Liste
- $::$ angewendet auf einen Wert und eine Liste, ergibt wieder eine Liste

⇒ Funktionen über Listen sind meist rekursiv

Länge einer Liste

List.tl liefert Liste außer erstem Element. Damit:

```
# let rec length l =  
  if l = []  
  then 0  
  else 1 + length (List.tl l);;  
val length : 'a list -> int = <fun>
```

```
# length ['A';'B'];;  
- : int = 2
```

Beob.: `length` ist polymorph im Elementtyp der Liste

Pattern-Matching mit match

Bequeme Möglichkeit, Datenstruktur auseinanderzunehmen

Syntax: `match expr with case | case | ...`

wobei `case = pattern -> expr`

`pattern` kann Variablen enthalten

Damit

```
# let rec length lst =  
  match lst with  
    [] -> 0  
  | head :: tail -> 1 + (length tail);;
```

Mögliche Patterns

- Variablen
- Ignore (`_`)
- Konstanten (selten, außer `[]`)
- Konstruktor mit Argumenten:
 - `::` für Listen
 - `(,)` für Paare
 - `(, . . .)` für Tupel
 - Später mehr

Komplexeres Beispiel

```
# let rec insert elt lst =
  match lst with
  | [] -> [elt]
  | head :: tail ->
    if elt <= head
    then elt :: lst
    else head :: insert elt tail

let rec sort lst =
  match lst with
  | [] -> []
  | head :: tail -> insert head (sort tail);;

val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
```

Funktionen als Objekte erster Klasse

Funktionen lassen sich als Parameter übergeben

```
# let deriv f dx x = (f(x +. dx) -. f(x)) /. dx;;  
val deriv : (float -> float) ->  
            float -> float -> float = <fun>
```

```
# deriv sin 1e-6 pi;;  
- : float = -1.00000000013961143
```

oder in Datenstrukturen speichern:

```
# [sin;cos;tan];;  
- : (float -> float) list = [<fun>; <fun>; <fun>]
```

Higher-Order Funktionen

Funktion auf alle Elemente einer Liste anwenden

```
# let rec map f l =  
    match l with  
    | [] -> []  
    | hd :: tl -> f hd :: map f tl;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map square [0;1;2;3;4];;  
- : int list = [0; 1; 4; 9; 16]
```

Listen filtern

Prädikat bestimmt, welche Elemente drin bleiben dürfen

```
# let rec filter p l =  
  match l with  
  | [] -> []  
  | hd::tl ->  
    if p hd  
    then hd::(filter p tl)  
    else filter p tl;;  
val filter :  
  ('a -> bool) -> 'a list -> 'a list = <fun>  
  
# filter even [1;2;3;4];;  
- : int list = [2; 4]
```

Anonyme Funktionen

Ausdruck, der zu Funktion ausgewertet: `fun var ... -> expr`

```
# fun x -> x + 1;;  
- : int -> int = <fun>
```

```
# map (fun x -> x + 1) [0;1;2;3;4];;  
- : int list = [1; 2; 3; 4; 5]
```

```
# let compose f g = fun x -> f(g(x));;  
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b  
              = <fun>
```

```
# let cos2 = compose (fun x -> x *. x) cos;;  
val cos2 : float -> float = <fun>
```

Records

Datentyp mit benannten Feldern

Syntax: `type name = { field ; ... }`

Wobei `field ::= name : type`

```
# type ratio = {num: int; denum: int};;
type ratio = { num: int; denum: int }
```

Records erzeugen: `{ name = expr ; ... }`

```
# let add_ratio r1 r2 =
  {num = r1.num * r2.denum + r2.num * r1.denum;
   denum = r1.denum * r2.denum};;
val add_ratio : ratio -> ratio -> ratio
# let elevenfifteen = add_ratio {num=1; denum=3}
                                {num=2; denum=5};;
val elevenfifteen : ratio = {num = 11; denum = 15}
```

Recordfelder lesen

Auf Felder zugreifen: *expr . name*

```
# elevenfifteen.num;;
```

```
- : int = 11
```

Records mit veränderbaren Feldern

Alternative für `field ::= mutable name : type`

```
# type colleague = {name: string; mutable married: bool}
type colleague = { name : string; mutable married : bool }
# let mandeep = {name="Mandeep"; married = false};;
val mandeep : colleague = {name = "Mandeep"; married = false}
```

Verändern: `expr <- expr`

```
# mandeep.married <- true;;
- : unit = ()
# mandeep;;
- : colleague = {name = "Mandeep"; married = true}
```

= und ==

Beide Operatoren vergleichen zwei beliebige Werte gleichen Typs

Ausnahme: Funktionen

Unterschied: = testet auf strukturelle Gleichheit == testet auf physische Gleichheit

Umgangssprachlich:

- $a = b \Rightarrow a$ ist gleich b
- $a == b \Rightarrow a$ ist dasselbe wie b

= und ==

Läßt sich beobachten, wenn Dinge verändert werden können:

Wirkt sich die Veränderung an **a** auf **b** aus?

```
# let new_mandeeep = {name="Mandeeep"; married = true}
# mandeeep = newMandeeep;;
- : bool = true
# mandeeep == newMandeeep;;
- : bool = false
```

Algebraische Datentypen

Typen, deren Werte von unterschiedlichen Konstruktoren erzeugt werden

Konstruktornamen müssen mit Großbuchstabe beginnen

Syntax: `type name = constr | ...`

Wobei `constr ::= Name of type ...`

```
# type number = Int of int | Float of float | Error;;
type number = | Int of int | Float of float | Error
# Int 42;;
- : number = Int 42
# type sign = Positive | Negative;;
type sign = Positive | Negative
# let sign_int n = if n >= 0 then Positive
                    else Negative;;
val sign_int : int -> sign = <fun>
```

Funktionen auf algebraischen Datentypen

Zugriff stets mit `match`

```
# let add_num n1 n2 =
  match (n1, n2) with
    (Int i1, Int i2) ->
      (* Check for overflow of integer addition *)
      if sign_int i1 = sign_int i2 &&
         sign_int(i1 + i2) <> sign_int i1
      then Float(float i1 +. float i2)
      else Int(i1 + i2)
  | (Int i1, Float f2) -> Float(float i1 +. f2)
  | (Float f1, Int i2) -> Float(f1 +. float i2)
  | (Float f1, Float f2) -> Float(f1 +. f2)
  | (Error, _) -> Error
  | (_, Error) -> Error;;

val add_num : number -> number -> number = <fun>
```

```
# add_num (Int 1) (Float 2.2);;
```

```
- : number = Float 3 2
```

Polymorphe algebraische Datentypen

Datentyp über Typen parametrisiert

Syntax: `type name var .. = constr | ...`

constr verwendet Variablen

```
# type 'a option = Some of 'a  
| None;;
```

mapfilter

```
# let rec mapfilter p l =  
...
```

Rekursive algebraische Datentypen

Definition kann den definierten Typ verwenden

```
# type 'a btree = Empty
                | Node of 'a * 'a btree * 'a btree;;
type 'a btree = | Empty | Node of 'a * 'a btree * 'a b
```

Funktionen darüber sind dann i.d.R. rekursiv

```
# let rec depth btree =
    match btree with
    | Empty -> 0
    | Node(y, left, right) ->
        1 + max (depth left) (depth right)
val depth : 'a btree -> int = <fun>
# depth (Node (42, Node (23, Empty, Empty), Empty))
- : int = 2
```

Lokale Bindungen

Syntax: `let var = expr in expr`

```
# let f x =  
    let x = x + 1 in  
    let y = x + 1 in  
    let z = x + y in  
    z;;  
# f 1;;  
- : int = 5
```

Analog: `let rec ... and ... in expr`

Ausnahme-Behandlung

Exception definieren: `exception Name`

```
# exception Empty_list;;  
exception Empty_list
```

Exception werfen: `raise Name`

```
# let head l =  
    match l with  
    [] -> raise Empty_list  
  | hd :: tl -> hd;;  
val head : 'a list -> 'a = <fun>  
# head [];;
```

Uncaught exception: `Empty_list`

Exception abfangen: `try expr with Name -> expr`

```
# let name_of_binary_digit digit =  
    try  
        List.assoc digit [0, "zero"; 1, "one"]
```

Module

Jede Datei ist ein eigenes Modul

Name des Moduls = Dateiname ohne `.m1` und erster Buchstabe groß

Namen sind nur im Modul sichtbar

Module können einzeln kompiliert werden

System ist eigentlich viel mächtiger

Zugriff auf andere Module

Expliziter Zugriff: *Modulname . Name*

Implizit: `open` *Modulname* macht die Namen des Moduls verfügbar

Schnittstellen

Zu jedem Modul gehört eine Schnittstelle

Schnittstelle bestimmt, welche Namen von außen sichtbar sind

Default-Schnittstelle: Alle Namen sichtbar

Explizite Schnittstelle: `.mli`-Datei mit Exporten

Export:

- `val name : type`
- Typdefinition
- Exceptiondefinition

Compilieren von Modulen

Datei `foo.ml`

```
let bar = 1
```

Zu Modul `Foo` kompilieren:

```
ocamlc -c foo.ml
```

⇒ `foo.cmo`

Compilieren von Schnittstellen

Datei `foo.mli`

```
val bar : int
```

Kompilieren:

```
ocamlc -c foo.mli
```

⇒ `foo.cmi`

Laden

```
ocaml foo.cmo
```

oder interaktiv

```
# #load "foo.cmo";;
```