Funktionale Programmierung

Spartakiade 2015 *Carsten König*

Einführung

kurze Einführung über die Slides

Tools

- Visual Studio
- F#-Interactive
- .FS vs .FSX
- F# PowerTools
- Ordnerstruktur / Reihenfolge
- Emacs / fsharp-mode

Grundlagen

Werte und Typen

`let` mit ein paar Zahlen Beispiel:

```
let antwort = 42;;
antwort / 2 + 2;;
```

Typen und Typ-Synonyme mit `type`

```
type Zahl = int
```

generische Typen

```
type Mit<'a> = 'a
type 'a Mit  = 'a
let x : _ Mit = 5
```

UNIT

- statt `void` benutzen wir `unit`
- `unit` hat nur einen Wert `()`
- spart die lästige Unterscheidung Funktion/Prozedur

BOTTOM als *nicht-totale* Sprache haben wir immer einen zustäzlichen Wert *bottom*:

```
failwith "bottom"
```

Funktionen

einfache Definition und Typen

```
let plus10 x = x + 10;;
plus10 5;;
```

Applikation hat höchste Priorität

```
plus10 5 * 5
```

Einrückungsregeln

```
let plus10 x =
    x + 10
```

Funktionen als Werte / innere Funktionen

```
let plus10 x =
    let plus5 x =
        x+5
    plus5 x + plus5 x
```

Lambdas

```
let plus10 =
    fun x -> x + 10
```

Komposition / Verkettung

```
let f x = x + x
let g x = string x

g (f 5) = (g << f) 4 = (f >> g) 4

4 |> f |> g
g <| (f <| 4)
```

mehrere Argumente / Currying

```
let f x y = x + y
f 3 4;;
```

betrachte Typ!

```
(f 3) 4 = 7
4 |> f 3 = 7
```

Erkläre Currying mit `int -> int -> int` = `int -> (int -> int)` **Flipchart**

Vorsicht! Applikation assoziiert nach links

```
let hi s = "Hallo " + s;;
hi System.DateTime.Now.ToString();;
```

QUIZ Sind folgende Funktionen gleich?

```
let f a b c = a + b + c
let g a = fun b c -> a + b + c
```

Merksatz:

> *Typ-Signaturen* sind **recht**-assoziiert, *Funktions-Applikation* ist **links**-assoziiert

partielle Applikation

```
let plus10 = f 10
```

erklären.

Als weiteres Beispiel: `printfn` und co.

Rekursion

```
let rec fact n =
    if n = 1 then 1 else
    n * fact (n-1)
```

Erkläre `rec` und `if`

ÜBUNG FizzBuzz Lasse die Teilnehmer *FizzBuzz* implementieren

---

Lösung

```
let fizzBuzz n =
    match n with
    | _ when n % 15 = 0 -> "FizzBuzz"
    | _ when n % 5  = 0 -> "Buzz"
    | _ when n % 3  = 0 -> "Fizz"
    | _                 -> string n
```

Strukturen und Patternmatching

Tupel

```
let t3 = (1, "Hallo", 3.5)
```

```
let (i,s,f) = t3
```

geht auch mit Konstanten und catchall

```
let (1,s,_) = t3
```

`select case` on steroids:

```
let test t =
  match t3 with
  | (1,s,_) -> "mit 1 " + s
  | (0,s,f) -> "mit 0 " + s + " und " + string f
  | _       -> "else/default"
```

QUIZ

1. gebe ein Beispiel für ein Tupel vom Type

- `bool * unit * char` an (`false, (), 'c'`)
- `string * (int * int) * bool` (`"Hi", (5,2), true`)

2. Schreibe Funktionen

- `first : 'a*'b -> 'a` und `second : 'a*'b -> 'b`
- `curry : ('a*'b -> 'c) -> ('a -> 'b -> 'c)`
- `uncurry : ('a -> 'b -> 'c) -> ('a*'b -> 'c)`

Listen und Sequenzen

```
let l = [1; 2; 3; 4; 5]
let l = [1..5]
let s = seq [1; 2; 3; 4]
let s = seq [1..5]
```

Head/Tail/Cons

```
let l = 1 :: [2..5]
let l = 1 :: 2 :: 3 :: 4 :: 5 :: []

let (x::xs) = l

let f = function
    | [] -> "Leer"
    | (x::xs) -> sprintf "Head: %a, Tail: %A" x xs
```

Concat

```
[1..5] = [1;2;3] @ [4;5]
```

QUIZ

- Schreibe Funktionen `head : 'a list -> 'a` und `tail : 'a list -> 'a list`
- Schreibe eine Funktion, die das 3. Element einer Liste liefert
- Implementiere Dein eigenes *Concat*

Lösung:

```
let rec concat xs ys =
    match xs with
    | [] -> ys
    | (x::xs) -> x :: concatxs ys
```

ÜBUNG Lasse die Teilnehmer das CoinChange implementieren

Lösung

```
let rec findChange remCoins remAmount =
    if remAmount = 0 then [] else
    match remCoins with
        | [] ->
            failwith "ich kann darauf nicht zurückgeben"
        | (c::_) when c <= remAmount ->
            c :: findChange remCoins (remAmount - c)
        | (_::cs) ->
            findChange cs remAmount

let coinChange (amount : int) : Coin list =
    coins = List.sort coins |> List.rev
    findChange coins amount
```

DUs und algebraische Datentypen Wie *Enumerationen*

```
type Enumeration = A | B | C
```

jeder *Fall* darf aber Werte enthalten (muss aber nicht)

```
type T = A of int | B of string | C
```

```
let a = A 5
let b = B "Hallo"
let c = C
```

`A`, `B` und `C` heißen *Daten-Konstruktoren* (es gibt auch *Typ-Konstruktoren* die in F# keine entsprechung haben)

Warum *algebraisch*? Fragen: - Wieviele Elemente hat `bool * bool`? - Wieviele Elemente hat `type T = A of bool | B of bool*bool | C`?

pattern - matching

```
let (A i) = a

match t with
    | A i -> string i
    | B s -> s
    | C   -> "leer"
```

dürfen rekursiv sein

```
type Expr =
    | Zahl of int
    | Plus of Expr * Expr
```

QUIZ

- gib einen Typ an, der entweder ein `int*int` Tupel oder nichts enthält
- schreibe eine Funktion `eval :  Expr -> int`

Option Ein oft verwendeter Typ

```
type 'a Option = None | Some 'a
```

der Ersatz für das lästige `Null`

ÜBUNG Lasse die Teilnehmer das Lösen quadratischer Gleichungen implementieren

---

Lösung

```
type Loesungen =
    | Keine
    | Alles
    | EineVon  of Loesung list
```

```
let loeseLinear (a : float, b : float) : Loesungen =
    if nahe0 a && nahe0 b then Alles
    elif nahe0 a then Keine
    else EineVon [-b / a]

let loese ((a,b,c) : QuadGleichung) : Loesungen =
    if nahe0 a then loeseLinear (b,c) else
    let nenner = 2.0*a
    let diskriminante = b*b - 4.0*a*c
    match diskriminante with
        | d when nahe0 d -> EineVon [-b / nenner]
        | d when d < 0.0 -> Keine
        | d ->
            let wurzel = sqrt d
            EineVon [(-b-wurzel)/nenner; (-b+wurzel)/nenner]
```

Records

Listen falten

Im Prinzip kann das alles als Übung freigegeben werden.

Länge einer Liste

```
let rec laenge = function
    | []      -> 0
    | (_::xs) -> 1 + laenge xs
```

Summe

```
let rec summe = function
    | []      -> 0
    | (x::xs) -> x + summe xs
```

Produkt

```
let rec produkt = function
    | []      -> 1
    | (x::xs) -> x * produkt xs
```

Map

gegeben: `f : 'a -> 'b` und `'a list` gesucht: `'b list`

```
let rec map f = function
    | []      -> []
    | (x::xs) -> f x :: map f xs
```

Filter

gegeben: Prädikat `p : 'a -> bool` und `ls : 'a list` gesucht: `'a list` mit Elementen `l` aus `ls` mit `p l = true`

```
let rec filter p = function
    | []      -> []
    | (x::xs) ->
        let xs' = filter p xs
        if p x then x :: xs' else xs'
```

Muster gesehen?

Muster herausarbeiten...

```
let rec foldR f s = function
    | []      -> s
    | (x::xs) -> f x (foldR f s xs)
```

Zeigen wofür das **R** in *foldR* steht

Merkregel

`foldR` ersetzt `[]` mit `s` und `::` mit `‘f‘`

Nochmal..

Alle Funktionen von oben nochmal mit `foldR` implementieren.

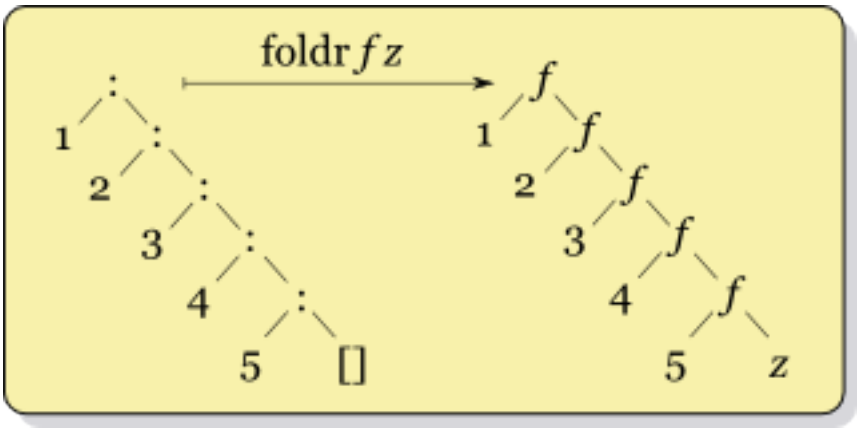

Figure 1: foldR

QUIZ

Zusätzlich:

- `and : bool list -> bool`
- `or : bool list -> bool`
- `any : ('a -> bool) -> 'a list -> bool`
- `all : ('a -> bool) -> 'a list -> bool`

implementiern.

Diskussion: Problem? (kürzt nicht ab)

in Haskell: Kein Problem wegen lazy - ist aber sowieso schon alles im `List` bzw. `Seq` Modul enthalten.

aber: es kann sich lohnen mit `Seq` statt `List` zu arbeiten!

Left-Fold

Analog zu `foldR`:

```
let rec foldL f s = function
    | []      -> s
    | (x::xs) -> foldL f (f s x) xs
```

- als `List.fold` enthalten

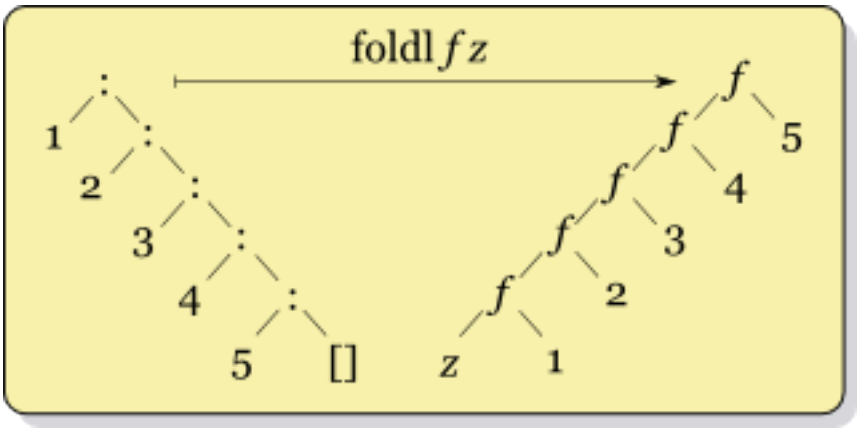

Figure 2: foldL

- `foldR` heißt `List.foldBack` (ein wenig anders definiert)

Vorteile:

- *tail-recursive* (Erklären)
- deswegen vorzuziehen

hartes QUIZ

Definiere `foldL` durch `foldR` (ohne direkte Rekursion)

```
let foldL f s xs =
    (foldR (fun x g s -> g (f s x)) id xs) s
```

Sequenzen, rekursion und Kombinatorik

Sequenz der Teillisten

Aufschreiben und erklären

```
let rec sublists = function
    | [] -> Seq.singleton []
    | (x::xs) -> seq {
        for xs' in sublists xs do
```

```
            yield xs'
            yield x::xs'
        }
```

crossProd

Erklären und implementieren lassen

```
let rec crossProd = function
    | []        -> Seq.singleton []
    | (xs::xss) ->
        let xss' = crossProd xss
        seq {
            for x in xs do
            for xs in xss' do
            yield (x::xs)
        }
```

Sequenz der Permutationen

Versuchen als Übung - Tipp mit `selectOne` geben:

```
let rec selectOne (xs : 'a list) : ('a * 'a list) seq =
    match xs with
    | [] -> Seq.empty
    | (x::xs) ->
        seq {
            yield (x,xs)
            for (x',xs') in selectOne xs do
                yield (x', x::xs')
        }

let rec permutationen (xs : 'a list) : ('a list) seq =
    match xs with
    | [] -> Seq.singleton []
    | _ ->
        seq {
            for (x,xs) in selectOne xs do
            for xs' in permutationen xs do
            yield x::xs'
        }
```

Unfold (?)

```
let rec unfold next start =
    seq {
        match next start with
        | None ->
            yield! Seq.empty
        | Some (v, n) ->
            yield v
            yield! unfold next n
    }
```

Iterate mit Unfold

```
let iterate f =
    Seq.unfold (fun s -> Some (s, f s))
```

Map mit Unfold

```
let map f =
    Seq.unfold (function
    | []      -> None
    | (x::xs) -> Some (f x, xs))
```

Zip mit Unfold

```
let zip xs ys =
    Seq.unfold
        (function
            | ([],_) | (_,[]) -> None
            | (x::xs,y::ys)   -> Some ((x,y),(xs,ys))
        )
        (xs,ys)
```

Bemerkungen: - `List.zip`: beide Listen müssen die gleiche Länge haben - `Seq.zip`: wie oben: bricht ab, wenn eine Sequenz leer wird - Umwandlung zwischen `List` und `Seq` mit `Seq.toList`, `Seq.ofList`, ...

Übung

Pascals Dreieck

```
let pascalSeq : int list seq =
    let gen row =
        let next =
            Seq.zip (0::row) (row@[0])
            |> List.ofSeq
            |> List.map (fun (a,b) -> a+b)
        Some (row, next)
    Seq.unfold gen [1]
```

Sudoku Projekt

Stelle das Sudoku Projekt vor und gehe es Schritt für Schritt durch

---

Funktoren

Einführung in kat. Theorie bis Funktoren

Die üblichen Beispiele:

- Listen
- Option
- `'a ->`

Monaden-Beispiele

Beispiele:

- Listen
- Option (mit Builder)
- Ws-Monade have fun
- Async Workflows
- event. async

Reise nach T²DD

Gemeinsames Spiel: ich empfehle Bowling Kata