Funktionale Programmierung in F# (3)

Grundlagen & Funktionales Design

Göran Kirchner[1]

2022-04-08

[1] e_kirchnerg@doz.hwr-berlin.de

Programm

- Vertiefung Railway-Oriented Programming
- Prinzipien des funktionalen Designs
- Refactoring (Übung)

Übung 1

- Implementiere einen Workflow (`validateInput`).

```
type Input = {Name : string; Email : string }
let checkNameNotBlank input =
  if input.Name = "" then
     Error "Name must not be blank"
  else Ok input
let checkName50 input =
  if input.Name.Length > 50 then
     Error "Name must not be longer than 50 chars"
  else Ok input
let checkEmailNotBlank input =
  if input.Email = "" then
     Error "Email must not be blank"
  else Ok input
```

```
type Input =
  {
    Name: string
    Email: string
  }
```

Übung 1 (Lösung)

```
let validateInput input =
    input
    |> checkNameNotBlank
    |> Result.bind checkName50
    |> Result.bind checkEmailNotBlank

let goodInput = {Name="Max"; Email="x@example.com"}
let blankName = {Name=""; Email="x@example.com"}
let blankEmail = {Name="Nora"; Email=""}
[validateInput goodInput; validateInput blankName; validateInput
↪  blankEmail]
```

```
val validateInput: input: Input -> Result<Input,string>
val goodInput: Input = { Name = "Max"
                         Email = "x@example.com" }
val blankName: Input = { Name = ""
                         Email = "x@example.com" }
val blankEmail: Input = { Name = "Nora"
                          Email = "" }
val it: Result<Input,string> list =
```

Übung 2

- Definiere einen *Custom Error Type*. Benutze diesen in den Validierungen.
- Übersetze die Fehlermeldungen (EN, FR, DE?).

```
type ErrorMessage =
  | ??   // name not blank
  | ?? of int  // name not longer than
  | ??   // email not longer than
let translateError_EN err =
  match err with
  | ?? -> "Name must not be blank"
  | ?? i -> sprintf "Name must not be longer than %i chars" i
  | ?? -> "Email must not be blank"
  | SmtpServerError msg -> sprintf "SmtpServerError [%s]" msg
```

Übung 2 (Lösung)

```
type ErrorMessage =
    | NameMustNotBeBlank
    | NameMustNotBeLongerThan of int
    | EmailMustNotBeBlank
    | SmtpServerError of string
let translateError_FR err =
    match err with
    | NameMustNotBeBlank -> "Nom ne doit pas être vide"
    | NameMustNotBeLongerThan i -> sprintf "Nom ne doit pas être plus
↪  long que %i caractères" i
    | EmailMustNotBeBlank -> "Email doit pas être vide"
    | SmtpServerError msg -> sprintf "SmtpServerError [%s]" msg
```

Funktionales Design

⇝ Functional Design Patterns

Prinzipien (1)

- Funktionen sind Daten!
- überall Verkettung (Composition)
- überall Funktionen
- Typen sind keine Klassen
- Typen kann man ebenfalls verknüpfen (algebraische Datentypen)
- Typsignaturen lügen nicht!
- statische Typen zur Modellierung der Domäne (später mehr;)

Prinzipien (2)

- Parametrisiere alles!
- Typsignaturen sind "Interfaces"
- Partielle Anwendung ist "Dependency Injection"
- Monaden entsprechen dem Chaining of Continuations"
  - bind für Options
  - bind für Fehler
  - bind für Tasks
- map Funktionen
  - Nutze map Funktion von generische Typen!
  - wenn man einen generischen Typ definiert, dann auch eine map Funktion

Übung 3

- Typsignaturen
- Funktionen sind Daten

Übung 4 (Think of a Number)

```fsharp
let thinkOfANumber numberYouThoughtOf =
    let addOne x = x + 1
    let squareIt x = ??
    let subtractOne x = ??
    let divideByTheNumberYouFirstThoughtOf x = ??
    let subtractTheNumberYouFirstThoughtOf x = ??

    // define these functions
    // then combine them using piping

    numberYouThoughtOf
    |> ??
    |> ??
    |> ??
```

Übung 4 (Lösung)

```
let thinkOfANumber numberYouThoughtOf =
    let addOne x = x + 1
    let squareIt x = x * x
    let subtractOne x = x - 1
    let divideByTheNumberYouFirstThoughtOf x = x / numberYouThoughtOf
    let subtractTheNumberYouFirstThoughtOf x = x - numberYouThoughtOf
    numberYouThoughtOf
    |> addOne
    |> squareIt
    |> subtractOne
    |> divideByTheNumberYouFirstThoughtOf
    |> subtractTheNumberYouFirstThoughtOf
thinkOfANumber 42
```

```
val thinkOfANumber: numberYouThoughtOf: int -> int
val it: int = 2
```

Übung 5 (Decorator)

- Implementiere das Decorator-Emtwurfsmuster für `add1`.

Pause

> If we'd asked the customers what they wanted, they would have said "faster horses".
>
> – Henry Ford

Tree Building (Übung)

```
exercism download --exercise=tree-building --track=fsharp
```

Tree Building (Imperativ)

```
let buildTree records =
    let records' = List.sortBy (fun x -> x.RecordId) records
    if List.isEmpty records' then failwith "Empty input"
    else
        let root = records'.[0]
        if (root.ParentId = 0 |> not) then
            failwith "Root node is invalid"
        else
            if (root.RecordId = 0 |> not) then failwith "Root node is
            ↪  invalid"
            else
                let mutable prev = -1
                let mutable leafs = []
                for r in records' do
                    if (r.RecordId <> 0 && (r.ParentId > r.RecordId ||
                    ↪  r.ParentId = r.RecordId)) then
                        failwith "Nodes with invalid parents"
                    else
                        if r.RecordId <> prev + 1 then
                            failwith "Non-continuous list"
```

Tree Building (Funktional)

```
let buildTree records =
    records
    |> List.sortBy (fun r -> r.RecordId)
    |> validate
    |> List.tail
    |> List.groupBy (fun r -> r.ParentId)
    |> Map.ofList
    |> makeTree 0

let rec makeTree id map =
    match map |> Map.tryFind id with
    | None -> Leaf id
    | Some list -> Branch (id,
        list |> List.map (fun r -> makeTree r.RecordId map))
```

Tree Building (Error Handling)

```
let validate records =
    match records with
    | [] -> failwith "Input must be non-empty"
    | x :: _ when x.RecordId <> 0 ->
        failwith "Root must have id 0"
    | x :: _ when x.ParentId <> 0 ->
        failwith "Root node must have parent id 0"
    | _ :: xs when xs |> List.exists (fun r -> r.RecordId < r.ParentId)
    ↪  ->
        failwith "ParentId should be less than RecordId"
    | _ :: xs when xs |> List.exists (fun r -> r.RecordId = r.ParentId)
    ↪  ->
        failwith "ParentId cannot be the RecordId except for the root
        ↪  node."
    | rs when (rs |> List.map (fun r -> r.RecordId) |> List.max) >
    ↪  (List.length rs - 1) ->
        failwith "Ids must be continuous"
    | _ -> records
```

Tree Building (Benchmarking)

- BenchmarkDotNet

```
dotnet run -c release
```

```
sed -n 447,460p $benchmarks
```

// * Summary *
BenchmarkDotNet=v0.12.1, OS=macOS 12.3 (21E230) [Darwin 21.4.0] Intel Core i7-7920HQ CPU 3.10GHz (Kaby Lake), 1 CPU, 8 logical and 4 physical cores .NET Core SDK=6.0.200 [Host] : .NET Core 6.0.2 (CoreCLR 6.0.222.6406, CoreFX 6.0.222.6406), X64 RyuJIT DEBUG DefaultJob : .NET Core 6.0.2 (CoreCLR 6.0.222.6406, CoreFX 6.0.222.6406), X64 RyuJIT

Method	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Alloc
Baseline	8.058 s	0.1562 s	0.1535 s	8.069 s	1.00	0.00	3.3569	-	-	13.75
Mine	5.172 s	0.2075 s	0.5953 s	5.006 s	0.57	0.02	1.8768	-	-	7.68

Zusammenfassung

- funktionaler Umgang mit Fehlern (ROP)
- funktionales Design
- funktionales Refactoring

Links

- fsharp.org
- docs.microsoft.com/../dotnet/fsharp
- F# weekly
- fsharpforfunandprofit.com
- github.com/../awesome-fsharp

Hausaufgabe (Erinnerung)

- exercism.io (E-Mail bis 22.04)
  - ☐ Queen Attack
  - ☐ Raindrops
  - ☐ Gigasecond
  - ☐ Bank Account
  - ☐ Accumulate
  - ☐ Space Age
- exercism.io (E-Mail bis 29.04)
  - ☐ Poker (Programmieraufgabe)