Elm

Reaktive Programmierung

Christopher Karow, Johannes Brauer

21. 08. 2021

Ziel

- Erstes Kennenlernen von Elm
- Beispiele teilweise entnommen aus Beginning Elm
- Literaturbeispiel: [Fel20]

Elm . . .

. . .

- ist strikt funktional
- besitzt ein (sehr) strenges, statisches und implizites Typsystem
- kennt keinen Null-Wert
- kennt keine Laufzeitfehler
- wird nach Javascript übersetzt
- liefert außerordentlich detaillierte Fehlermeldungen
- dient (nur) zur Programmierung von Web-Anwendungen
- benötigt dafür kein Framework
- ist „auf natürliche Weise" reaktiv

Die Sprache

Werte

- Es gibt die Zahlentypen `Int`, `Float`
  - und die Abstraktion `number` (constrained type variable)

```
> 1 + 2
3 : number

> 4534 - 789
3745 : number

> 42 * 10
420 : number

> 9 / 4
2.25 : Float
```

```
> 5 / 2
2.5 : Float

> 5 // 2
2 : Int

> 5 / 3
1.6666666666666667 : Float

> (+) 4 5
9 : number

> 2 ^ 3 ^ 2
512 : number
```

  - Es gelten die üblichen Vorrangregeln.

- Es gibt Booleans `True` und `False`

```
> True || False
True

> False || False
False

> True && False
False

> True && True
True

> not True
False

> xor True False
True

> xor True True
False

> xor False False
False
```

  - Nur `True` und `False` werden als Booleans akzeptiert.

- Es gibt Zeichen (`Char`) und Zeichenketten (`String`)

```
> "Reaktive Programmierung"
"Reaktive Programmierung" : String

> "Reaktive" ++ " Programmierung"
"Reaktive Programmierung" : String
> 'R'
'R' : Char
```

Konstanten (aka Variablen)

```
> x = (7 + 5) / 3
4 : Float
> x
```

```
4 : Float

> anzahlDerTeilnehmerInnenAmWahlplichtmodulReaktiveProgrammierung = 9
9 : number

> anzahlDerTeilnehmerInnenAmWahlplichtmodulReaktiveProgrammierung // 3
3 : Int

> x = x + 1
-- CYCLIC DEFINITION ------------------------------------------------- REPL

The `x` value is defined directly in terms of itself, causing an infinite loop.

3| x = x + 1
   ^
Are you are trying to mutate a variable? Elm does not have mutation, so when I
see x defined in terms of x, I treat it as a recursive definition. Try giving
the new value a new name!

Maybe you DO want a recursive value? To define x we need to know what x is, so
let's expand it. Wait, but now we need to know what x is, so let's expand it...
This will keep going infinitely!

Hint: The root problem is often a typo in some variable name, but I recommend
reading <https://elm-lang.org/0.19.1/bad-recursion> for more detailed advice,
especially if you actually do need a recursive value.
```

Funktionen

- Funktionsdefinitionen sehen ähnlich aus wie Konstantendefinitionen

```
> add x y = x + y
<function> : number -> number -> number

> add
<function> : number -> number -> number

> add 2 3
5 : number

-- partielle Anwendung
> add 2
<function> : number -> number

> (add 2) 6
8 : number

> add2 = add 2
<function> : number -> number

> add2 9
11 : number
```

- Nutzung der Pipe-Operatoren

```
> add x y = x + y
<function> : number -> number -> number

> increment = add 1
<function> : number -> number
```

```
-- der Operator |> übergibt den linken Ausdruck an die rechts stehende Funktion
> ten = 9 |> increment

-- der Operator <| übergibt den rechten Ausdruck an die links stehende Funktion
> increment <| add 5 4
10 : number
```

- Lambda-Ausdrücke
  - Syntax: `\argumente -> berechnungsvorschrift`

```
normalFunction x y = x + y
-- ist äquivalent zu
lambdaFunction = \x y -> x + y

> normalFunction 1 2
3 : number

> lambdaFunction 1 2
3 : number
```

- lokale Variablen

```
bigNumbers =
    let
        allNumbers = List.range 1 100
        isBig num = num > 95
    in
        List.filter isBig allNumbers

> bigNumbers
[96,97,98,99,100] : List Int
```

Fallunterscheidungen

- `if`

```
> velocity = 11.2
11.2

> if velocity > 11.186 then "Godspeed" else "Come back"
"Godspeed"
+ =else= muss immer da sein.
```

- `case`

  statt

```
weekday dayInNumber =
    if dayInNumber == 0 then
        "Sunday"

    else if dayInNumber == 1 then
        "Monday"

    else if dayInNumber == 2 then
        "Tuesday"

    else if dayInNumber == 3 then
        "Wednesday"
```

```
    else if dayInNumber == 4 then
        "Thursday"

    else if dayInNumber == 5 then
        "Friday"

    else if dayInNumber == 6 then
        "Saturday"

    else
        "Unknown day"
```

besser:

```
weekday dayInNumber =
    case dayInNumber of
        0 ->
            "Sunday"

        1 ->
            "Monday"

        2 ->
            "Tuesday"

        3 ->
            "Wednesday"

        4 ->
            "Thursday"

        5 ->
            "Friday"

        6 ->
            "Saturday"

        _ ->
            "Unknown day"
```

Strukturen

Listen

- Syntax: `[value1,value2,value3.....valuen]`
- alle Elemente müssen vom gleichen Typ sein
- Beispiele

```
> myList1 = [10,20,30]
[10,20,30] : List number

> myList2 = ["hello","world"]
["hello","world"] : List String

> List.head [10,20,30,40] --aka first
Just 10 : Maybe number
> List.head []
Nothing : Maybe a
```

```
> List.tail [10,20,30,40] -- aka rest
Just [20,30,40] : Maybe (List number)
> List.tail []
Nothing : Maybe (List a)

10::[20,30,40,50]  -- aka cons
[10,20,30,40,50] : List number

> List.isEmpty
<function> : List a -> Bool -- a ist eine Typvariable

> List.isEmpty [10,20,30]
False : Bool

> List.product [4, 5, 6]
120 : number
> List.product []
1 : number

> List.maximum
<function> : List comparable -> Maybe comparable
> List.maximum [3, 17, 2]
Just 17 : Maybe number
> List.maximum []
Nothing : Maybe comparable

> List.filter
<function> : (a -> Bool) -> List a -> List a
List.filter (\n -> (remainderBy 2 n) == 0) [10,20,30,55]

-- und so weiter ...
```

Arrays

- vergleichbar mit Vektoren in Clojure
- auf die Elemente kann per Index zugegriffen werden

```
> import Array
> myArray = Array.fromList [ 1, 2, 3, 4 ]
Array.fromList [1,2,3,4]
    : Array.Array number

> Array.get 3 myArray
Just 4 : Maybe number
> Array.get 4 myArray
Nothing : Maybe
```

Tuples

- Strukturen, die aus zwei oder drei Elementen (möglicherweise) unterschiedlichen Typs bestehen

```
( 1, "abc" )
(1,"abc") : ( number, String )

> ( 1, "abc", ( "Weltformel", 42 ))
(1,"abc",("Weltformel",42))
    : ( number, String, ( String, number1 ) )

> trianglePerimeter ( a, b, c ) = a + b + c
```

```
<function> : ( number, number, number ) -> number
-- Funktion erwartet 1 Argument, Destructuring

> trianglePerimeter ( 5, 4, 6 )
15 : number
```

Records

- Sammlung von Schlüssel-Wert-Paaren

```
> { x = 3, y = 4 }
    : { x : number, y : number1 }

> point3D =
  { x = 3
  , y = 4
  , z = 12
  }
{ x = 3, y = 4, z = 12 }
    : { x : number, y : number1, z : number2 }

> daniel = { name = "Düsentrieb", alter = 85 }
{ alter = 85, name = "Düsentrieb" }
    : { alter : number, name : String }

> point3D.z
12 : number
> .z point3D
12 : number
> .age daniel
85 : number

-- Pattern matching / Destructuring

> hypotenuse {x,y} =
    sqrt (x^2 + y^2)
<function> : { a | x : Float, y : Float } -> Float

> hypotenuse point3D
5 : Float

> hypotenuse { x = 3, y = 4 }
5 : Float

-- Updating

> { point3D | x = 0, y = 0 }
{ x = 0, y = 0, z = 12 }
    : { x : number, y : number1, z : number2 }
```

Typen

Typinferenz

```
> toFullName person = person.firstName ++ " " ++ person.lastName
<function> : { a | firstName : String, lastName : String } -> String

> fullName = toFullName { firstName = "Gustav", lastName = "Gans" }
"Gustav Gans" : String
```

```
> fullName = toFullName { firstName = "Donald", lstName = "Duck" }
-- TYPE MISMATCH ------------------------------------------------------------ REPL

The 1st argument to `toFullName` is not what I expect:

14| fullName = toFullName { firstName = "Gustav", lstName = "Gans" }
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

This argument is a record of type:

    { firstName : String, lstName : String }

But `toFullName` needs the 1st argument to be:

    { a | firstName : String, lastName : String }

Hint: Seems like a record field typo. Maybe lastName should be lstName?
```

Typ-Annotationen

- zur Verbesserung von Fehlermeldungen
- zur Dokumentation

```
hypotenuse : Float -> Float -> Float
hypotenuse a b =
  sqrt (a^2 + b^2)
```

Benennung von Typen

```
type alias Person =
    { name : String
    , alter : Int
    }

stimmberechtigt : Person -> Bool
stimmberechtigt person =
    person.alter >= 18

> stimmberechtigt { name = "Clarissa", alter = 20}
True : Bool

-- ohne type alias
stimmberechtigt : { name : String, alter : Int} -> Bool
stimmberechtigt person =
    person.alter >= 18

-- Typvariablen für Record-Typen dienen auch als Konstruktoren
> Person "Clarissa" 20
{ alter = 20, name = "Clarissa" } : Person
```

Typdefinitionen

```
> type Gruss = Hallo | Moin

> Hallo
Hallo : Gruss
> Moin
Moin : Gruss

type Bool = False | True
```

```
gruesse : Gruss -> String
gruesse gruss =
    case gruss of
        Hallo ->
            "Tach zusammen"
        Moin ->
            "Moin Moin"

> gruesse
<function> : Gruss -> String
> gruesse Hallo
"Tach zusammen" : String
```

Typvariablen

- In den folgenden Beispielen ist `a` eine Typvariable.

```
> List.append
<function> : List a -> List a -> List a

> type Maybe a = Just a | Nothing

> Just 5
Just 5 : Maybe number

> Nothing
Nothing : Maybe a

divide : Int -> Int -> Maybe Int
divide dvd dvs =
    case dvs of
        0 -> Nothing
        _   -> Just (dvd // dvs)

> divide
<function> : Int -> Int -> Maybe Int

> divide 8 7
Just 1 : Maybe Int
> divide 8 0
Nothing : Maybe Int
```

Einfache Web-Seite mit Elm

```
bash-3.2$ elm init

Hello! Elm projects always start with an elm.json file. I can create them!

Now you may be wondering, what will be in this file? How do I add Elm files to
my project? How do I see it in the browser? How will my code grow? Do I need
more directories? What about tests? Etc.

Check out <https://elm-lang.org/0.19.1/init> for all the answers!

Knowing all that, would you like me to create an elm.json file now?
[Y/n]: Y

Okay, I created it. Now read that link!
```

```
bash-3.2$ l
total 8
drwxr-xr-x@  4 jb  staff  128 Aug 19 13:57 ./
drwxr-xr-x@ 20 jb  staff  640 Aug 19 13:55 ../
-rw-r--r--@  1 jb  staff  517 Aug 19 13:57 elm.json
drwxr-xr-x@  2 jb  staff   64 Aug 19 13:57 src/
bash-3.2$ l src
total 0
drwxr-xr-x@ 2 jb  staff   64 Aug 19 13:57 ./
drwxr-xr-x@ 4 jb  staff  128 Aug 19 13:57 ../

bash-3.2$ elm make src/ReactiveProgramming.elm --output elm.js
Starting downloads...

   elm/json 1.1.3
   elm/browser 1.0.2
   elm/core 1.0.5

Dependencies ready!
Success! Compiled 1 module.

    ReactiveProgramming > elm.js
```

ReactiveProgramming.elm

```
module ReactiveProgramming exposing (main)

import Html exposing (..)
import Html.Attributes exposing (..)


view model =
    div []
        [div [ class "myCss" ]
             [ h1 [] [ text "Willkommen im Modul Reaktive Programmierung" ]
             , p [] [ text "Dozenten: Chhristopher Karow, Johannes Brauer"
            ]
        ],
         h3 [] [text "Teilnehmer*Innen"]]


main =
    view "dummy model"
```

repr.html

Die Elm-Architecture

- Drei Bestandteile
  - Model: beinhaltet den Weltzustand
  - View: verwandelt den Weltzustand in HTML
  - Update: berechnet einen neuen Weltzustand als Reaktion auf Nachrichten

Elementares Beispiel: Counter
weitere Beispiele in `https://elm-lang.org/try`

[width=.9]./Abbildungen/architecture

Abbildung 1: entnommen aus `https://guide.elm-lang.org/architecture/`

Literaturverzeichnis

[Fel20] Richard Feldman. *Elm in Action*. Manning Publications Co., 2020.