Unser Pitch zur funktionalen Programmierung

mittels F#

Christian Parpart & Kei Thoma

Humboldt Universität zu Berlin

27. Juni 2019

$$i = i + 1$$

Wir werden nicht:

1. eine neue Programmiersprache lernen
2. funktionale Programmieren lernen

Unser Pitch zur funktionalen Programmierung

mittels F#

Christian Parpart & Kei Thoma

Humboldt Universität zu Berlin

27. Juni 2019

Inhalt

Immutability

Python

```
a = 0

i = 1
i = i + 1
```

F#

```
let a = 0

let mutable i = 1
i <- i + 1
```

Notation

Mathe

$$a = 1$$
$$f : x \mapsto x + a$$

$$g(f(x))$$
$$(g \circ f)(x)$$

F#

```
let a      = 1
let f (x) = x + a
```

```
g(f(x))
```

```
(g << f)(x)
```

Seiteneffektfreiheit

Code

```
let f(x) =
    let a = 2
    let b = 3
    a * b + x
```

Eine Funktion, die nur von ihren Argumenten abhängt, ist seiteneffektfrei!

Wir gewinnen die folgenden Garantien:

1. Vorherbestimmtheit (Determinismus)
2. Vereinfachte Code Verständlichkeit
3. Vereinfachte Refaktorisierung

Wiederverwendbarkeit

- folgt aus der Seiteneffektfreiheit
- nicht aller Code ist 25 Zeilen lang
  - rechts im Bild: 1023 Zeilen
  - links im Bild: 15894 Zeilen: Seite zu klein ;-(

Verifizierbarkeit

- Theorem Proving und Mutable Variablen (Z3, CVC4)
- FP Programme sind leichter zu verifizieren

Fazit

- Probleme von Funktionaler Programmierung
  - Schwierig sich rein zu denken
  - Langsamer als C/C++
    (aber immer noch schneller als Python)
  - wenn es einmal läuft, wie ein Stein
- Fun Fact: F# steht für ___

Fazit

- Probleme von Funktionaler Programmierung
  - Schwierig sich rein zu denken
  - Langsamer als C/C++
    (aber immer noch schneller als Python)
  - wenn es einmal läuft, wie ein Stein
- Fun Fact: F# steht für FUN

Bonusmaterial

```fsharp
// Classic iterative (imperative) implementation.
let prime_factors (n: int): int list =
    let mutable result: int list = []
    let mutable n': int = n
    for i in 2 :: [ 3 .. 2 .. n / 2 ] do
        while (n' % i) = 0 do
            result <- i :: result
            n' <- n' / i
    result
```

Bonusmaterial

```
let prime_factors (n: int): int list =
    let rec append_and_divide (result: int list) (n': int) (i: int): int list * int =
        match (n' % i) <> 0 with
        | true -> (result, n')
        | false -> append_and_divide (i :: result) (n' / i) i
    let rec factorize (acc: int list, n': int) (i: int option): int list =
        let next_i = function
            | i when i > n / 2 -> None
            | 2 -> Some (3)
            | i -> Some (i + 2)
        match i with
        | Some i' -> factorize (append_and_divide acc n' i') (next_i i')
        | None -> acc
    factorize ([], n) (Some 2)
```

Bonusmaterial

```
let inline private rowCanonicalStep (d: int) (work: State< ^F>) : State< ^F> =
    let rec step (d: int) (work: State< ^F>) : State< ^F> =
        let mat = matrixState work
        if d > min (rowCount mat) (columnCount mat) then
            work
        else
            ensureValueOneAt work d d
            |> makeZerosBelow d d
            |> makeZerosAbove d d
            |> step (d + 1)
    step d work
```