Functional programming introduction

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Course introduction

- This course is about *functional programming* (FP)
- FP is a programming paradigm that focuses on data transformation instead of memory manipulation

Introduction

Other paradigms

- Imperative programming (JavaScript, Java)
- OO programming (Java)
- Declarative programming (SQL)

Introduction

Imperative programming

- Most used, most important programming tool nowadays
- Strongly connected to hardware
- CPU and memory are inherently imperative

Introduction

Imperative programming

- Mostly focused on the notion of *destructively updating* **state**
- `x = y;` loses (destroys) the previous value of `x`

Introduction

The risks of state

- State and references cause undesired behaviours
- Each thread/function/class has pointers to shared state
- Each thread/function/class makes locally innocent modifications that as a whole break everything

```
money = 10000eur

putMoney m =
  newMoney := money + m
  money    := newMoney

takeMoney m =
  if money > m then
    newMoney := money - m
    money    := newMoney
    return OK
  else
    return NOT_ENOUGH_BALANCE
```

Introduction

The risks of state

- `runParallel(takeMoney 10000, takeMoney 10000)` may return OK

```
for a in asteroids
  if collision(a, projectiles)
    remove(a, asteroids)

for p in projectiles
  if collision(p, asteroids)
    remove(p, projectiles)
```

Introduction

The risks of state

- Will never remove anything from `projectiles`!

```
static missileConnection = connect("192.168.1.1//nuke"
    )

class EmployeePayroll =
  DoNotPayProgrammerBonuses : () -> () =
    missileConnection.Launch()
```

Introduction

The risks of state

- Will cause WW3 if programmer bonuses are not paid

Introduction

The risks of state

- Concurrent updates are dangerous
- Creation of "half processed" states of the program which make no sense

Introduction

The risks of state

- Functions that return `void` may really do anything
- A function signature is meaningless
- We cannot prevent wrong composition of functions with libraries

Introduction

Referential transparency

- One of the core tenets of FP
- Same input always yields same output
- Predictable code

Introduction

Referential transparency

- How do we get this?
- We forbid...

Introduction

Referential transparency

- How do we get this?
- We forbid...
- MUTABLE VALUES!!!!!

Introduction

Referential transparency

- Easier testing (only input values, not function order)
- Stronger encapsulation (less dependencies from order)
- Less chances of access mistakes (no way to access unrelated stuff)

Introduction

Introduction to ML

- FP is very old
- First high-level programming language, in the 50's, was LISP (FP)
- Never been particularly popular
  - It makes programming more difficult
  - It *seems* to yield better code on average

Introduction

Introduction to ML

- We will use a dialect of ML (Meta-Language)
- Early 70's
- Hybrid programming language which discourages, but does not forbid, value mutation

Introduction

Introduction to ML

- We will use F#
- Originally built by Microsoft
- Now fully open sourced
- Best debugger, tool, and library support of any other FP language
- Most other FP languages have "academic quality" (that is not very high) of tooling

```
let (i:int) = 100

let (x:float) = 10.0

let (b:bool) = true
```

```
let f (i:int) (j:int) : int = i + j
```

```
let quadratic (a:float) (b:float) (c:float) (x:float)
    : float =
  let t0 = c
  let t1 = x * b
  let t2 = x * x * a
  t0 + t1 + t2
```

```
let distance (x1:float,y1:float) (x2:float,y2:float) :
    float =
  let dx = x1-x2
  let dy = y1-y2
  System.Math.Sqrt(dx * dx + dy * dy)
```

```
let tooCloseToZero (p:float * float) : bool =
  if distance (0.0,0.0) p < 10.0 then
    true
  else
    false
```

```
let someNumbers (i:int) : string =
  match i with
  | 0 -> "zero"
  | 1 -> "one"
  | 3 -> "three"
  | 5 -> "five"
  | 7 -> "seven"
  | _ -> "number␣I␣don't␣like"
```

```
let rec fac (n:int) : int =
  if n = 0 then
    1
  else
    n * fac(n-1)
```

```
let rec fac (n:int) : int =
  match n with
  | 0 -> 1
  | _ -> n * fac(n-1)
```

```
let rec fac : int -> int =
  function
  | 0 -> 1
  | n -> n * fac(n-1)
```

Introduction

Conclusions and assignment

- FP is a powerful programming paradigm
- Very different from imperative programming
- Functions are just “data pumps”
  - Data goes in
  - No side-effects
  - Data goes out

Introduction

Conclusions and assignment

- The assignments are on Natschool
- **Restore** the games to a working state
- Hand-in a **printed** report that only contains your **sources** and the associated **documentation**

Introduction

Conclusions and assignment

- Any book on the topic will do
- I did write my own (Friendly F#) that I will be loosely following for the course, **but it is absolutely not mandatory or necessary to pass the course**

Dit is het

The best of luck, and thanks for the attention!