Functional Programming

Marco Patrignani

K.U.Leuven

19 October 2012
Goal of the Lecture

Goal of the Lecture

- remind/illustrate FP style, and related concepts;

Goal of the Lecture

- remind/illustrate FP style, and related concepts;
- remind why FP is important and its strong points.\(^1\)

---

Goal of the Lecture

- remind/illustrate FP style, and related concepts;
- remind why FP is important and its strong points.\(^1\)

- Introduce you to the syntax of Racket.

---

Style of the Lecture

INTERACTIVE
Style of the Lecture

INTERACTIVE

With explanation on the side.
Organisation of the Lecture

- 1/2 (or more): FP
Organisation of the Lecture

- 1/2 (or more): FP
- 1/2 (or less): Racket syntax
Functional Programming: Motivation
Functional Programming: Motivation

- modularisation;
Functional Programming: Motivation

- modularisation;

- "no" side effects.
Motivation

Outline
Motivation 1: Modular Code

*Divide et impera*
Motivation 1: Modular Code

*Divide et impera:* break a problem in subproblems and *glue* them to solve the problem.
Motivation 1: Modular Code

*Divide et impera*: break a problem in subproblems and *glue* them to solve the problem.

**HOW?**
Motivation 1: Modular Code

*Divide et impera*: break a problem in subproblems and *glue* them to solve the problem.

**HOW?** (what is the glue?)
Motivation 1: Modular Code

*Divide et impera:* break a problem in subproblems and *glue* them to solve the problem.

**HOW?** (what is the glue?)

- algebraic data types (seen yesterday);
Motivation 1: Modular Code

*Divide et impera*: break a problem in subproblems and *glue* them to solve the problem.

**HOW?** (what is the glue?)

- algebraic data types (seen yesterday);
- functions;
Motivation 1: Modular Code

*Divide et impera*: break a problem in subproblems and *glue* them to solve the problem.

**HOW?** (what is the glue?)

- algebraic data types (seen yesterday);
- functions;
- functions as first-class citizens (higher-order functions);
Motivation 1: Modular Code

*Divide et impera:* break a problem in subproblems and *glue* them to solve the problem.

**HOW?** *(what is the glue?)*

- algebraic data types (seen yesterday);
- functions;
- functions as first-class citizens (higher-order functions);
- laziness.
Motivation 1: Modular Code

WHY?
Motivation 1: Modular Code

**WHY?**

- code is *smaller* (strangely a good thing);
Motivation 1: Modular Code

WHY?

- code is *smaller* (strangely a good thing);
- code can be reasoned about *in isolation*; (also, verified)
Motivation 1: Modular Code

WHY?

- code is *smaller* (strangely a good thing);
- code can be reasoned about *in isolation*; (also, verified)
- code can be reused.
Motivation 1: Modular Code

WHY?

- code is *smaller* (strangely a good thing);
- code can be reasoned about *in isolation*; (also, verified)
- code can be reused. (similar motivation for OO programming! what is the difference?)

Homework
Think about this!
Motivation 2: “No” Side Effects

Side effects are:

- modify the value of a global/static variable;
- exceptions;
- input/output operations;

Maybe we need them.
Motivation 2: “No” Side Effects

Side effects are:

- modify the value of a global/static variable;
Side effects are:

- modify the value of a global/static variable; (well, who cares)
Motivation 2: “No” Side Effects

Side effects are:
- modify the value of a global/static variable; (well, who cares)
- exceptions;
Motivation 2: “No” Side Effects

Side effects are:
- modify the value of a global/static variable; (well, who cares)
- exceptions; (mmm)
Motivation 2: “No” Side Effects

Side effects are:

- modify the value of a global/static variable; (well, who cares)
- exceptions; (mmm)
- input/output operations;
Motivation 2: “No” Side Effects

Side effects are:

- modify the value of a global/static variable; (well, who cares)
- exceptions; (mmm)
- input/output operations; (ouch!)
Motivation 2: “No” Side Effects

Side effects are:
- modify the value of a global/static variable; (well, who cares)
- exceptions; (mmm)
- input/output operations; (ouch!)

Maybe we need them.
Motivation 2: “No” Side Effects

**Pros**

- variables do not change value;
  (no state)
Motivation 2: “No” Side Effects

**Pros**

- variables do not change value; (no state)
- functions only compute their result;
Motivation 2: “No” Side Effects

**Pros**

- variables do not change value; (no state)
- functions only compute their result;
- irrelevant order of execution!

Achievable via Monads (more on Dave’s lectures).
Motivation 2: “No” Side Effects

**Pros**

- variables do not change value;  
  (no state)
- functions only compute their result;
- irrelevant order of execution!
- parallelisable code!!
Motivation

### Motivation 2: “No” Side Effects

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- variables do not change value; (no state)</td>
<td>- variables do not change value; (no state)</td>
</tr>
<tr>
<td>- functions only compute their result;</td>
<td></td>
</tr>
<tr>
<td>- irrelevant order of execution!</td>
<td></td>
</tr>
<tr>
<td>- parallelisable code!!</td>
<td></td>
</tr>
</tbody>
</table>

Achievable via Monads (more on Dave’s lectures).
Motivation 2: “No” Side Effects

**Pros**
- variables do not change value; (no state)
- functions only compute their result;
- irrelevant order of execution!
- parallelisable code!!

**Cons**
- variables do not change value; (no state)
- no I/O, seriously?
Motivation 2: “No” Side Effects

**Pros**
- Variables do not change value; (no state)
- Functions only compute their result;
- Irrelevant order of execution!
- Parallelisable code!!

**Cons**
- Variables do not change value; (no state)
- No I/O, seriously?

Achievable via Monads (more on Dave’s lectures).
Outline
Algebraic data types are kind of a composite type (Nat in the Scala assignment) based on the *induction principle*.
Algebraic data types are kind of a composite type (Nat in the Scala assignment) based on the *induction principle*.

**Induction**

- provide a Base case;
- provide a way to construct elements based on “smaller” elements. (Inductive case)
Algebraic data types are kind of a composite type (Nat in the Scala assignment) based on the induction principle.

**Induction**

- provide a Base case;
- provide a way to construct elements based on “smaller” elements. (Inductive case)

Pattern matching is
Algebraic data types are kind of a composite type (Nat in the Scala assignment) based on the *induction principle*.

**Induction**

- provide a Base case;
- provide a way to construct elements based on “smaller” elements. (Inductive case)

Pattern matching is

you tell me! you were told yesterday!
Coding time #1

Natural numbers, double, addition, multiplication, equality, maximum.
Functions as First-class Citizens

Functions as First-class Citizens

??

Entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.

What do we do with functions?

**define them;**

**call them;**

**send them as parameters!**
Functions as First-class Citizens

Entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.
Functions as First-class Citizens

What do we do with functions?

Entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.
Functions as First-class Citizens

Entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.

What do we do with functions?

- define them;
- call them;
- send them as parameters!
Functions as First-class Citizens

Entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.

What do we do with functions?

- define them;
- call them;
- send them as parameters!
Extra reasoning when defining a function: **TYPES**.
Functions as First-class Citizens

Functions with Types

Extra reasoning when defining a function: **TYPES**.

- A function that inputs a Nat and outputs a Nat is written:
  \[ \text{Nat} \rightarrow \text{Nat} \]
Functions with Types

Extra reasoning when defining a function: **TYPES**.

- A function that inputs a Nat and outputs a Nat is written:
  \[ \text{Nat} \rightarrow \text{Nat} \]

- A function that inputs *two* Nat’s and outputs a Nat is written:
  \[ \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \]

The *type checker* ensures that the program is *well-typed* (in a typed programming language).
Functions with Types

Extra reasoning when defining a function: **TYPES**.

- A function that inputs a Nat and outputs a Nat is written:
  \[ \text{Nat} \rightarrow \text{Nat} \]

- A function that inputs *two* Nat’s and outputs a Nat is written:
  \[ \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \]

The *type checker* ensures that the program is *well-typed* (in a typed programming language).

This is lame...
Functions with *Polimorphic* Types

A function that inputs *any* type and returns *that* type:

\[ \forall a. a \rightarrow a \]
Functions with *Polimorphic* Types

- A function that inputs *any* type and returns *that* type:
  \[ \forall a. a \rightarrow a \]

This is the ***!
Functions with *Polimorphic* Types

- A function that inputs *any* type and returns *that* type:
  \[ \forall a. a \rightarrow a \]

This is the ****!

**AND**

The type checker can infer it *automatically* (more or less, in certain languages).
Coding time #2
Lists, sum of elements of a list, tail recursive sum, length, append.
Functions with *Better* Types

- A function that inputs an element of type $a$, a function from any type $a$ to another type $b$ and outputs something of that type $b$ is written:

$$\forall a, b. \ a \rightarrow (a \rightarrow b) \rightarrow b$$
A function that inputs an element of type $a$, a function from any type $a$ to another type $b$ and outputs something of that type $b$ is written:

$$\forall a, b. \ a \to (a \to b) \to b$$

**Higher order!**
Higher-Order Functions

Coding time #3
Map, filter, foldr.
Outline
foldr's, foldr's everywhere!

foldr

The structure of a foldr is not new.
foldr's, foldr's everywhere!

The structure of a foldr is not new. Consider the list [1, 1, 1].
The structure of a foldr is not new. Consider the list [1, 1, 1].

```
foldr's, foldr's everywhere!
```

```
foldr
```

```
The structure of a foldr is not new. Consider the list [1, 1, 1].
```

```
cons
```

```
1
```

```
cons
```

```
1
```

```
cons
```

```
1
```

```
cons
```

```
1
```

```
il
```

```
∀ a. a → [a] → [a]
```

```
∀ a. [a]
```

```
The type of a foldr is:
```

```
∀ a, b. [a] → b → (a → b → b) → b
```

```
Marco Patrignani
```

```
K.U.Leuven
```

```
Functional Programming
```
foldr

The structure of a foldr is not new. Consider the list [1, 1, 1].

What is the type of cons?
foldr's, foldr's everywhere!

The structure of a foldr is not new. Consider the list \([1, 1, 1]\).

What is the type of \texttt{cons}?

\[
\forall a. \, a \rightarrow [a] \rightarrow [a]
\]

\[
\begin{array}{c}
\text{cons} \\
\text{cons} \\
1 \\
\text{cons} \\
1 \\
\text{cons} \\
1 \\
\text{nil}
\end{array}
\]
The structure of a `foldr` is not new. Consider the list `[1, 1, 1]`.

What is the type of `cons`?

\[ \forall a. \ a \to [a] \to [a] \]

And the type of `nil`?
foldr

The structure of a foldr is not new. Consider the list [1, 1, 1].

What is the type of cons?

$$\forall a. \ a \to \ [a] \to \ [a]$$

And the type of nil?

$$\forall a. [a]$$
foldr

The structure of a foldr is not new. Consider the list [1, 1, 1].

What is the type of cons?

$$\forall a. a \rightarrow [a] \rightarrow [a]$$

And the type of nil?

$$\forall a. [a]$$

The type of a foldr is:

$$\forall a, b. [a] \rightarrow b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b$$
foldr

The structure of a foldr is not new. Consider the list [1, 1, 1].

What is the type of cons?

\( \forall a. a \to [a] \to [a] \)

And the type of nil?

\( \forall a. [a] \)

The type of a foldr is:

\( \forall a, b. [a] \to b \to (a \to b \to b) \to b \)
The structure of a \texttt{foldr} is not new. Consider the list $[1, 1, 1]$.

What is the type of \texttt{cons}?

$\forall a. \ a \rightarrow [a] \rightarrow [a]$  

And the type of \texttt{nil}?

$\forall a. [a]$  

The type of a \texttt{foldr} is:

$\forall a, b. \ [a] \rightarrow b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b$
The structure of a \texttt{foldr} is not new. Consider the list \([1,1,1]\).

What is the type of \texttt{cons}?

$$\forall a. \ a \rightarrow [a] \rightarrow [a]$$

And the type of \texttt{nil}?

$$\forall a. [a]$$

The type of a \texttt{foldr} is:

$$\forall a, b. \ [a] \rightarrow b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b$$
foldr's, foldr's everywhere!

More Modularisation

Coding time #4

sum via foldr, append via foldr, length via foldr, map via foldr.
The operator \( \cdot \) allows for functions to be combined in the classical mathematical way.
The Type of Function Composition

The operator \( \cdot \) allows for functions to be combined in the classical mathematical way.

**Question:**

Given \( f : c \to a \) and \( g : b \to c \), what is the type of \( f \cdot g \) ?
The operator \( . \) allows for functions to be combined in the classical mathematical way.

**Question:**
Given \( f : c \to a \) and \( g : b \to c \), what is the type of \( f.g \)?

\( b \to a \)
More Modularisation

Coding time #4 bis

map via foldr.
Outline
More Complex Data Structures

What About ... ?

more complex data structures?
What About ... ?

more complex data structures?

Does the foldr scale on them?
What About ... ?

more complex data structures?

Does the foldr scale on them?

YES!

All inductively-defined data structures implicitly have a foldr. It is the concept of *catamorphism*. 
Binary Trees

Coding time #5

BTrees, foldr on trees, map on trees, map on trees via foldr, depth via foldr.
Laziness

What is laziness?
Laziness

- What is laziness?
- Why do we want/need laziness?
Laziness: What

Example: how do students study?
Laziness: What

Example: how do students study?

- delay the computation *until* you need it;
Laziness: What

Example: how do students study?

- delay the computation *until* you need it;
- “call by need” in the $\lambda$-calculus;
Laziness: Why

- to avoid large and possibly diverging computation;

\[(2 == 2) || (isPrime 997)\]
Laziness: Why

- to avoid large and possibly diverging computation;

(2 == 2) || (isPrime 997)

Similar to short-circuit evaluation, except *always*!
Laziness: Why

- to avoid large and possibly diverging computation;

\[(2 == 2) || (isPrime 997)\]

Similar to short-circuit evaluation, except *always*!
- to define infinite types (co-induction as opposed to induction);

```haskell
data Stream a = Elem a (Stream a)
```
Laziness: Why

- to avoid large and possibly diverging computation;

```
(2 == 2) || (isPrime 997)
```

Similar to short-circuit evaluation, except always!

- to define infinite types (co-induction as opposed to induction);

```
data Stream a = Elem a (Stream a)
```

Once upon a time ...
Laziness

Coding time #6

All numbers, all the even ones, all the prime ones.
download Racket, install it and open DrRacket;
download Racket, install it and open DrRacket;
above is the definitions area, below is the interaction one;
• download Racket, install it and open DrRacket;
• above is the definitions area, below is the interaction one;
• the first line defines the language you are using;
Syntax

Racket

donload Racket, install it and open DrRacket;
above is the *definitions* area, below is the *interaction one*;
the first line defines the language you are using;
write something in the interaction area;
download Racket, install it and open DrRacket;
above is the definitions area, below is the interaction one;
the first line defines the language you are using;
write something in the interaction area;
write something in the definition area and call it;
download Racket, install it and open DrRacket;
above is the definitions area, below is the interaction one;
the first line defines the language you are using;
write something in the interaction area;
write something in the definition area and call it;
Racket is: functional and untyped, so you can write functions that expect functions!
Racket

- conditional statements;
Syntax

Racket

- conditional statements;
- pattern-matching... on lists;
Racket

- conditional statements;
- pattern-matching... on lists;
- lambda functions.
Racket

- conditional statements;
- pattern-matching... on lists;
- lambda functions.

Play with it before the next lectures.