Fundamentals of Coq

Niels van der Weide — Radboud University, Nijmegen, The Netherlands
School on Univalent Mathematics
Minneapolis, July 2024

What is Coq?

- Coq is:
  1. a programming language;
  2. a proof assistant.

- In other words: Coq allows to write programs that build mathematical entities and formal proofs.

The rest of the School

UniMath = Coq − Ind + UA

This lecture!

UniMath is a **library** for Univalent Mathematics.

UniMath has been **developed on the proof assistant Coq, but it uses a slightly different type theory.**

Principal **differences** between plain Coq and UniMath (more will be said later):

- Certain features of Coq are rejected/not used (e.g, General (Co)Inductive Datatypes and other things).
- The Voevodsky's Axiom of Univalence (UA) is assumed.

Coq vs UniMath

UniMath is a library for Univalent Mathematics.
UniMath has been developed on top of the proof assistant Coq.

Roughly speaking, from the point-of-view of the logical systems:

$$
\begin{aligned}
\text{Coq} = \ & \text{Dependent Type Theory} \\
& + \text{Calculus of Constructions} \\
& + \text{(Co)Inductive constructions} \\
& + \ \dots
\end{aligned}
$$

$$
\begin{aligned}
\text{UniMath} = \ & \text{Coq} \\
& - \text{Calculus of Constructions} \\
& - \text{(Co)Inductive constructions} \\
& + \text{A basic collection of datatypes } (\mathscr{U}, \mathbb{N}, \text{bool}, \Pi, \Sigma, \mathrm{Id}, \dots) \\
& + \text{Univalence Axiom} \\
& \quad \dots
\end{aligned}
$$

Coq as a programming language

Your first program in Coq

Name of the program

Type of the program

```
Definition addtwo : nat -> nat :=
  λ n, n + 2.
```

Body of the program

This *program* takes a natural number n and returns $n + 2$.

Running a program

Coq source:

```
Definition addtwo : nat -> nat :=
  λ n, n + 2.

Eval compute in (addtwo 3).
```

Output:

```
5 : nat
```

Terminology

Using the terminology of Type Theory (see first lecture):

- *Programs* are called ***terms***.

  Use

  ```
  Definition ident : type := tm
  ```

  to bind the term *tm* to the constant *ident*.

- *Running* programs is called ***evaluation*** or ***normalization***.

  Use the command

  ```
  Eval compute in tm.
  ```

  to normalize the term *tm*.

Syntactic sugar for function definitions

- Functions are denoted using *λ-abstraction*:

```
Definition addtwo : nat -> nat :=
  λ n, n + 2.
```

- The λ-abstraction can be made implicit

```
Definition addtwo (n : nat) : nat :=
  n + 2.
```

- The two above snippets of code are equivalent.

Types

Basic examples of types

Type	Inhabitants	Description
`nat`	`0`, 1, 2, …	Natural numbers
`bool`	`true`, `false`	Booleans
`unit`	tt	Singleton
`empty`		Empty type
`dirprod A B`	`(x ,, y)`	Direct product (Cartesian product)
`coprod A B`	`ii1 a`, `ii2 b`	Coproduct (disjoint union)
`A -> B`	`fun` *var* `:` *ty* `=>` *body*	Function type
`UU`	`nat`, `bool`, `A -> B`, …	Universe (the type of types)

Terms and Types

Every term has an (univocally) associated type.

Examples:

- `(1 + 0) : nat`
- `true : bool`
- `(fun x:nat => x + 2) : nat -> nat`

Coq command `Check`

Use the command

Check *tm*.

to print the type of term *tm*.

Examples

Command:

```
Check (2 + 2).
```

Output:

```
: nat
```

Command:

```
Check true.
```

Output:

```
: bool
```

Command:

```
Check nat.
```

Output:

```
: UU
```

Notations

Special notations

Often two (or more) notations are available for certain mathematical expressions.

Examples:

Addition of natural numbers:

- `add m n` (basic syntax)
- `m + n` (alternative syntax)

Coproduct (disjoint sum) of types:

- `coprod A B` (basic syntax)
- `A ⊔ B` (alternative syntax)

Lambda expressions:

- `fun` *var* `=>` *body* (basic syntax)
- `λ` *var*`,` *body* (alternative syntax)

Frequently used notations

Basic syntax	Alt syntax	How to type	Description
add m n	m + n	+	Addition.
mul m n	m ∗ n	∗	Multiplication.
paths x y	x = y	=	Id-type (equality)
tpair x y	(x ,, y)	,,	Pair
dirprod A B	A × B	\times	Direct product (Cartesian product)
coprod A B	A ⨿ B	\amalg	Coproduct (Disjoint union)
fun v => b	λ x, b	\lambda	Lambda abstraction
A -> B	A → B	\to	Function type
empty	∅	\emptyset	Empty type

Π-types and Σ-types

Syntax for Π- and Σ-types

Given a type A and a type family

```
B : A -> UU
```

we have (see Lecture 1) the types $\prod_{a:A} B(a)$ and $\sum_{a:A} B(a)$

Basic syntax	Alternate syntax	How to type	Description
`forall (a:A), B a`	`Π (a:A), B a`	`\prod`	Dependent function type
`total2 (λ (a: A), B a)`	`Σ (a:A), B a`	`\sum`	Dependent pair type

Π-types

Example: identity function $A \to A$ for all types A

$$\texttt{idfun} : \prod_{A:\mathtt{UU}} (A \to A)$$

Definition in Coq:

```
Definition idfun : Π A : UU, A → A :=
    λ (A : UU) (a : A), a.
```

Functions with implicit arguments

In function application, certain arguments can be deduced by the context.

Example: Consider

In mathematical notation $I_{\mathbb{N}}(3)$

```
    idfun nat 3
```

the first argument (`nat`) can be deduced by the second one (3).

Arguments of a function can be declared implict using braces:

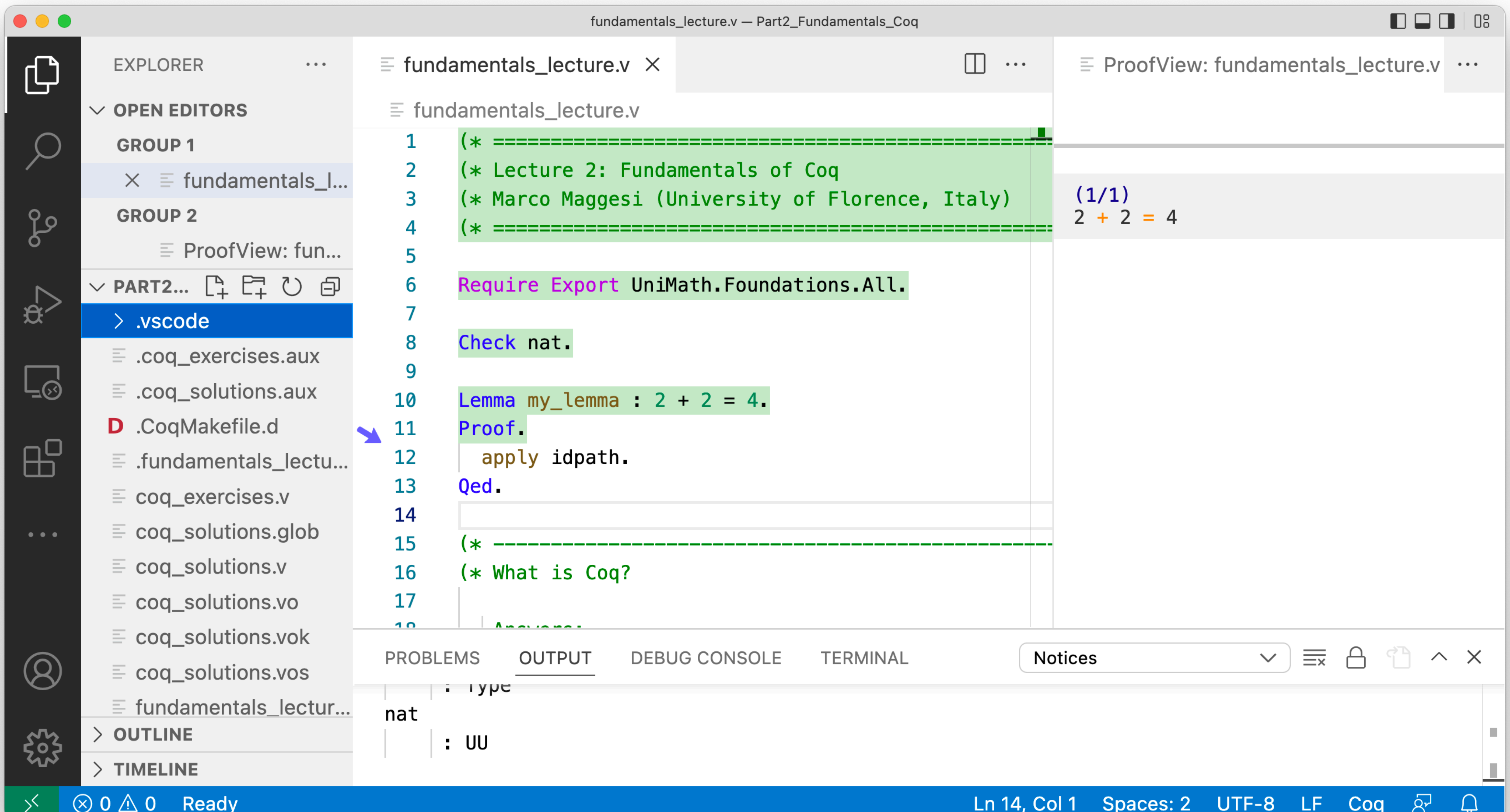
```
Definition idfun {A : UU} (a : A) : A := a.
```

Interacting with Coq

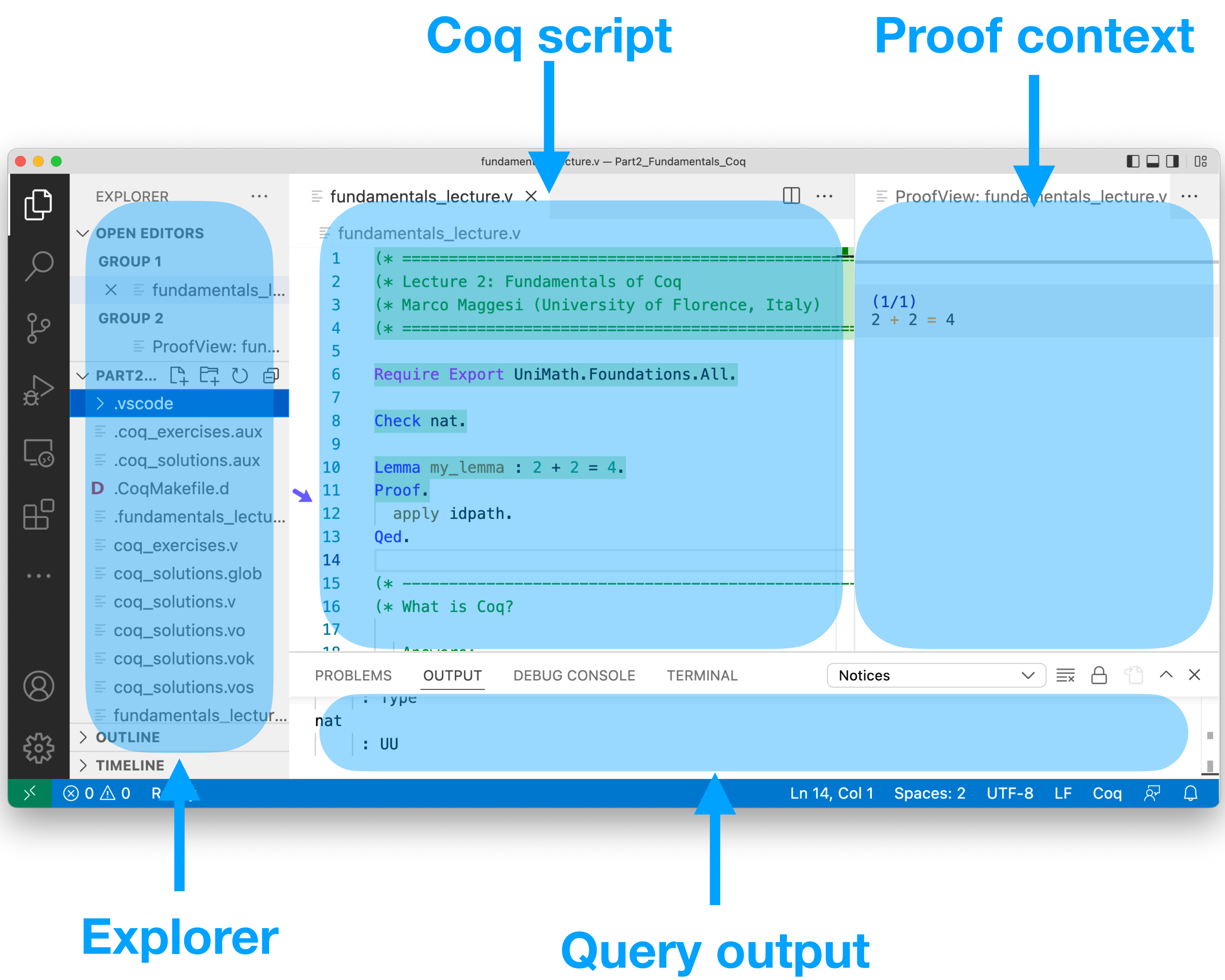
Our environment

- We will use Visual Studio Code or Codium to edit Coq scripts and to interact with Coq.
- Other options are available:
  - Emacs with Proof General
  - CoqIDE
- Contact us if you have problems setting up the environment on your computer.

Interacting with Coq in VScode



Interacting with Coq in VScode



Running Coq queries

Command	Linux & Win	Mac	Output
Check	Ctrl-Alt-C	^ ⌘ C	The type of a term
Print	Ctrl-Alt-P	^ ⌘ P	The definition of a constant
About	Ctrl-Alt-A	^ ⌘ A	Various information on an object (e.g. implicit arguments),
Locate	Ctrl-Alt-L	^ ⌘ L	Fully qualified name of an object or a special notation.