CS720
Logical Foundations of Computer Science
Lecture 1: course structure, Coq basics
Tiago Cogumbreiro
About the course

- **Classes:** Tuesday & Thursday
  12:30noon to 1:45pm at S-3-028
- **Office hours:** Tuesday & Thursday
  2:30pm to 4:00pm at S-3-088
- **Course web page:** piazza.com/umb/fall2018/cs720/home
Grading

- Homework: 75%
- Presentation: 15%
- Participation: 10%
Homework (75%)

- No late homework. Late homework = 0 points.
- Homework can be resubmitted up to one week. Final grade is the average of both submissions.
- Your lowest homework score will be dropped.
- Homework is your personal individual work.

It is acceptable to discuss the concept in general terms, but unacceptable to discuss specific solutions to any homework assignment.
Autograding

- Work is graded automatically. If Coq cannot check your homework, then your grade is 0 points.
- Use Admitted to allow for incomplete proofs and definitions.
Presentation (15%)

Choose one:

1. One chapter of the textbook to the class (60 minutes).
   The instructor will still publish the slides of that chapter.

2. One paper on the subject of formalizing the semantics of a programming language or system
   (20 minutes presentation).
   The student may suggest a paper (or request one suggestion).
Participation (10%)

Each reasonable student intervention (in the class or online) yields 1 point. If the student reaches 14 points, they are graded the full mark of participation.
Textbooks


Recommended

- **Types and programming languages.** Benjamin C. Pierce. 2002.
- **Software foundations @ YouTube**
- **Oregon PL Summer School Archives** (in particular: 2013, 2014, )
Suggested background

- **CS 450**: The Structure of Higher Level Languages
- **CS 451**: Compilers
Programming language semantics

- Describes a **computation model**
- Defines the set of possible behaviors through some primitives
- Mathematically precise properties of a computation model
Bird's eye view

Here is what we will learn
Imperative program

```ocaml
let division (a b: int) : int =
  let q = ref 0 in
  let r = ref a in
  while !r >= b do
    q := !q + 1;
    r := !r - b
  done;
  !q
```

Examples: OCaml, F#, ReasonML
Specifying a functional language

Language grammar

\[ t ::= x \mid v \mid t \, t \quad v ::= \lambda x : T . t \quad T ::= T \rightarrow T \mid \text{unit} \]

Evaluation rules

\[ \frac{t_1 \rightarrow t'_1}{t_1 \, t_2 \rightarrow t'_1 \, t_2} \quad (\text{E-app1}) \quad \frac{t_2 \rightarrow t'_2}{t_1 \, t_2 \rightarrow t_1 \, t'_2} \quad (\text{E-app2}) \]

\[ (\lambda x : T_{11}. t_{12}) \, v_2 \rightarrow [x \mapsto v_2] \, t_{12} \quad (\text{E-abs}) \]
Specifying a functional language

Type checking rules

\[
\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(T-var)}
\]

\[
\frac{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2}{\Gamma \vdash \lambda x : T \vdash t : T_2} \quad \text{(T-abs)}
\]

\[
\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \lambda x : T. t_2 : T_1 \rightarrow T_2} \quad \text{(T-app)}
\]
Mathematically precise properties

Progress

*Any valid program is either a value or can evaluate.*

If $\Gamma \vdash t : T$, then either $t$ is a value, or there exists some $t'$ such that $t \rightarrow t'$.

Subject reduction

*The validity of a program is preserved while evaluating it.*

If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

| Can you give an example of a property? |
Pre- and post-conditions

```plaintext
let division (a b: int) : int
    requires { true }
    ensures { exists r: int. a = b * result + r \ 0 ≤ r < b }
    =
    let q = ref 0 in
    let r = ref a in
    while !r ≥ b do
        invariant { true }
        q := !q + 1;
        r := !r - b
    done;
    !q
```

Examples: WhyML, Dafny.
What we will learn in this course

Course summary

**Specification:** logical reasoning, describing program behavior

**Abstraction:** capturing the fundamentals, thinking from first principles

**Testing:** unit and property testing
Part 1

A primer on the programming language Coq

We will learn the core principles behind Coq
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?

- boolean
- 4 suits of cards
- byte
- int32
- int64
Declare an enumerated type

\begin{verbatim}
Inductive day : Type :=
| monday : day
| tuesday : day
| wednesday : day
| thursday : day
| friday : day
| saturday : day
| sunday : day.
\end{verbatim}

- **Inductive** defines an (enumerated) type by cases.
- The type is named `day` and declared as a : `Type` (Line 1).
- Enumerated types are delimited by the assignment operator (`:=`) and a dot (`.`).
- Type `day` consists of 7 cases, each of which is is tagged with the type (`day`).
Printing to the standard output

*Compute* prints the result of an expression (terminated with dot):

```plaintext
Compute monday.
```

prints

```plaintext
  = tuesday
  : day
```
Interacting with the outside world

- Programming in Coq is different most popular programming paradigms
- Programming is an **interactive** development process
- The IDE is very helpful: workflow similar to using a debugger
- It’s a REPL on steroids!
- **Compute** evaluates an expression, similar to `printf`
Inspecting an enumerated type

```plaintext
match d with
| monday  => tuesday  
| tuesday  => wednesday
| wednesday => thursday
| thursday  => friday
| friday    => monday
| saturday  => monday
| sunday    => monday
end
```
Inspecting an enumerated type

```ocaml
match d with
| monday  => tuesday
| tuesday => wednesday
| wednesday => thursday
| thursday => friday
| friday  => monday
| saturday => monday
| sunday  => monday
end
```

- `match` performs **pattern matching** on variable `d`.
- Each pattern-match is called a *branch*; the branches are delimited by keywords `with` and `end`.
- Each *branch* is prefixed by a mid-bar (`|`) (→), a pattern (eg, `monday`), an arrow (→), and a return value.
Pattern matching example

```plaintext
Compute match monday with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.
```
Create a function

Definition next_weekday (d:day): day :=
match d with
| monday => tuesday
| tuesday => wednesday
| wednesday => thursday
| thursday => friday
| friday => monday
| saturday => monday
| sunday => monday
end.
Create a function

Definition next_weekday (d:day) : day :=
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.

- **Definition** is used to declare a function.
- In this case **next_weekday** has one parameter d of type day and returns (:) a value of type day.
- Between the assignment operator (:=) and the dot (.), we have the body of the function.
Example 2

Compute \( \text{next\_weekday friday} \).

yields (Message pane)

\[
\begin{align*}
\text{= monday} \\
\text{: day}
\end{align*}
\]

\text{next\_weekday friday} \text{ is the same as monday} \text{ (after evaluation)}
Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.

Proof.
  simpl. (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.
Your first proof

Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
  simpl. (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.

- Example prefixes the name of the proposition we want to prove.
- The return type (:) is a (logical) **proposition** stating that two values are equal (after evaluation).
- The body of function test_next_weekday uses the ltac proof language.
- The dot (.) after the type puts us in proof mode. (Read as "defined below".)
- This is essentially a unit test.
Ltac: Coq's proof language

Ltac is imperative! You can step through the state with CoqIDE

Proof begins an Ltac-scope, yielding

1 subgoal
______________________________________(1/1)
next_weekday (next_weekday saturday) = tuesday

Tactic simpl evaluates expressions in a goal (normalizes them)
Ltac: Coq's proof language

1 subgoal
______________________________________(1/1)
tuesday = tuesday

- reflexivity solves a goal with a pattern $?X = ?X$

No more subgoals.

- Qed ends an ltac-scope and ensures nothing is left to prove
Function types

Use `Check` to print the type of an expression:

```
Check next_weekday.
```

which outputs

```
next_weekday : day -> day
```

Function type `day -> day` takes one value of type `day` and returns a value of type `day`. 
Compound types

Enumerated types are very simple. You can think of them as a typed collection of constants. We call each enumerated value a constructor.

```
Inductive rgb : Type :=
| red : rgb
| green : rgb
| blue : rgb.
```
Enumerated types are very simple. You can think of them as a typed collection of constants. We call each enumerated value a **constructor**.

```plaintext
Inductive rgb : Type :=
  red : rgb
  green : rgb
  blue : rgb.
```

A **compound type** builds on other existing types. Their constructors accept *multiple parameters*, like functions do.

```plaintext
Inductive color : Type :=
  black : color
  white : color
  primary : rgb -> color.
```
Definition monochrome (c : color) : bool :=
   match c with
   | black => true
   | white => true
   | primary p => false
end.
Manipulating compound values

Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary p ⇒ false
end.

We can use the place-holder keyword _ to mean a variable we do not mean to use.

Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary _ ⇒ false
end.
Compound types

Allows you to: type-tag, fixed-number of values
Inductive types

How do we describe arbitrarily large/composed values?
Inductive types

How do we describe arbitrarily large/composed values?

Here's the definition of natural numbers, as found in the standard library:

```coq
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

- 0 is a constructor of type nat.
  *Think of the numeral 0.*

- If n is an expression of type nat, then S n is also an expression of type nat.
  *Think of expression n + 1.*

What's the difference between nat and uint32?
Recursive functions

Recursive functions are declared differently with **Fixpoint**, rather than **Definition**.

```coq
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S n') => evenb n'
end.
```

Using **Definition** instead of **Fixpoint** will throw the following error:

The reference evenb was not found in the current environment.

**Not all recursive functions can be described.** Coq has to understand that one value is getting "smaller."

**All functions must be total:** all inputs must produce one output. **All functions must terminate.**
Basic.v

- New syntax: **Definition** declares a non-recursive function
- New syntax: **Compute** evaluates an expression and outputs the result + type
- New syntax: **Check** prints the type of an expression
- New syntax: **Inductive** defines inductive data structures
- New syntax: **Fixpoint** declares a (possibly) recursive function
- New syntax: **match** performs pattern matching on a value
- New tactic: **simpl** evaluates functions if possible
- New tactic: **reflexivity** concludes a goal \(?X = ?X\)
Ltac vocabulary

- simpl
- reflexivity