SOFTWARE VERIFICATION AND COMPUTER PROOF (lesson 1)

Enrico Tassi – Inria Sophia-Antipolis

Who am I?

1. I'm a researcher at Inria
2. I work on proof assistants, the kind of tools that we will be using for this class. In particular Coq.
3. I've been doing so for quite a while, since I was your age actually, and I'm still having a lot of fun!

Who are you?

1. Who knows ML, OCaml, Haskell or Scala?
2. Who knows LISP, Scheme or Clojure?
3. Who knows Coq or another proof assistant?
4. Who knows what a type system is?
5. Who knows what structural recursion means?

What is this class about?

1. Formal methods: mathematics applied to software
2. Isn't testing software enough?
3. Our tool, Coq, provides a computer understandable (checkable) language to write programs and proofs

We learn to write programs and prove their correctness in Coq

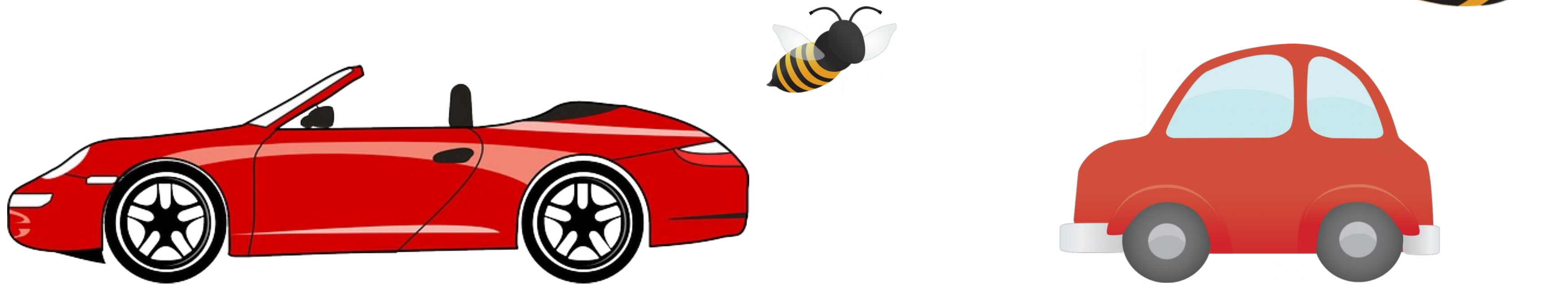
Lessons

1. Writing programs in Coq
2. Writing proofs in Coq
3. Proving your recursive programs correct
4. Proofs in arithmetic and more
5. More data types

Today: writing programs in Coq

1. Data types
2. Computation
3. Defining functions
4. Pattern matching
5. Recursive functions

What are types for?



- ***avoid confusion between totally different things***
  E.g. a bee is not a car, we can't drive it
- ***know something about similar things, but not everything***
  E.g. a car has an engine, wheels, ... and we can drive it. Still a Ferrari is not a Fiat 500.

Some “built in” types

- bool is the type of true and false
- nat is the type of 0, 1, 2, 3, ...
- Coq provides commands to check the type of an expression (e.g. Check 4)
- The answer 4 : nat has to be read ``4 has type nat''.
- We will often use (x : T) to annotate x with its type T

Some “built in” types and operations

- Operations on bool like conjunction (infix &&) and disjunction (infix ||) and negation (prefix negb)
- Operations on nat like addition (infix +) and subtraction (infix -) and multiplication (infix *)
- Coq provides commands to evaluate expressions and compute their result (Compute 3 + 7)

Little demo

Defining functions

- Definition myfunc (x : nat) := x + x.
- The type of functions is written with an arrow.
  Check myfunc prints nat → nat
- Anonymous functions
  Check (fun x : nat => x + x).
  Definition myfunc := (fun x : nat => x + x).

Calling functions, naming results

- Functions are applied by juxtaposition.

```
Check myfunc 4.
Compute myfunc 5.
```

- Value can be named locally.

```
Compute let ten := myfunc 5 in myfunc ten.
```

- Functions are first class.

```
Compute (let f := fun x : nat => x + x in f (f 3)).
```

A type error explained

- A sentence can be well formed but fail to type check
  Check (myfunc true).

  Error: The term "true" has type "bool" while it is expected to have type "nat".

Little demo

Syntactic sugar

- Coq provides a mechanism of notations
- We will not define new notations, but we must know a few things:
  - What is behind a notation: Locate "_ + _".
  - Notations are optional: Compute plus 2 3.
  - Also numbers are just sugar: Check (S (S O)).

More types

- The pair of any two types can be formed.
  E.g. (bool * nat) is the type of (true, 7) or (false, 2+5)
- The type of lists of elements of a type can be formed.
  E.g. (list nat) is the type of nil, 2::nil, 4::1::27::nil, ...
- The option of a type can represent success or failure.
  E.g. (option nat) is the type of (Some 7), None

Pattern matching (1/3)

- To inspect a value we can pattern match on it.
- E.g.: take the first component of a pair:

```
Definition fst (p : nat * bool) :=
 match p with
 | (n, _) => n
 end.
```

The wild card _ drops the unwanted part

Pattern matching (2/3)

- E.g.: Inspect a value of type list nat

```
Definition head (l : list nat) :=
 match l with
 | nil => None
 | x :: xs => Some x
 end.
```

All cases must be taken into account!

Pattern matching (3/3)

- E.g.: inspect a value of type nat

```
Definition is_zero (n : nat) :=
  match n with
  | O => true
  | S _ => false
  end.
```

- Look at the output of Print nat for the cases to be handled in the pattern matching

If then else

- It is just a match on values of type bool

```
Definition myif (b : bool) (a b : nat) :
  match b with
  | true => a
  | false => b
  end.
```

- There is actually some syntactic sugar for if then else

```
Check if true then 7 else 5.
```

Little demo

Recursion

- Compute the sum of the elements of a list:

```
Fixpoint slnat (l : list nat) :=
 match l with
 | nil => O
 | x :: xs => x + slnat xs
 end.
```

- We can run it: `Compute slnat (1 :: 3 :: 4 :: nil).`

Coq likes only *structural* recursion

- Recursive calls are allowed only on *smaller terms*
- Smaller terms can be obtained by pattern matching
- Fixpoint slnat (l : list nat) :=

```
  match l with
  | nil => ...
  | x :: xs => ... recursive call on xs ...
end.
```

- l is equal to x :: xs, hence xs is a sub term of l.
- Structurally recursive functions *do terminate!*

Non structural recursion: error explained

```
Fixpoint wrong (l : list nat) :=
  match l with
  | nil => O
  | x :: xs => x + wrong (17 :: nil)
  end.
```

Recursive call to wrong has principal argument equal to "17 :: nil" instead of "xs".

Another example: two inputs

- Test if a number n is smaller than m.

```
Fixpoint leb (n m : nat) :=
 match n, m with
 | O, _ => true
 | S _, O => false
 | S n1, S m1 => leb n1 m1
 end.
```

- Pattern match on both inputs at once (just sugar)
- Which is the principal argument?
- Which is the type of leb?

Last example: an higher order function

- Test if an element of the list validates p.

```
Fixpoint list_exists (p : nat → bool) (l : list nat) :=
 match l with
 | nil => false
 | x :: xs => (p x) || (list_exists p xs)
 end.
```

- Example of usage:

```
Compute (list_exists (fun x => leb x 17) (33 :: 4 :: nil).
```

That is all for today!