# Verified Extraction to OCaml from Coq, in Coq

Yannick Forster
Inria Team Gallinette, Nantes
Joint work with Matthieu Sozeau, Pierre Giraud, Pierre-Marie Pédrot, Nicolas Tabareau

received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101024493.

For example, misinterpretations of the law and inadequate translations into computer code can cause the software to function correctly with respect to its specifications, but disastrously with respect to the rule passed by the legislature or the executive branch.

# How to ensure that software functions correctly as specified?

# Different domains need different levels of correctness

- end-to-end encrypted messaging
- banking, finance, trading
- airplanes, cars, (self-driving) metros
- medical applications
- electronic voting
- basic infrastructure: power plants, electricity, internet infrastructure
- even mundane software: calendars, text processing, scanners
- applications in law

# How to ensure that software functions correctly as specified?

# Human testing

Answer: “I tested it. Then I got a colleague to test it. In the end, we even hired 10 people to work with the software for a week, and everything went great.”

In order to  believe in this process, you have to

- trust that the humans made no mistake
- trust that the humans checked every single edgecase

# Automated testing

Answer: “We have millions of automated tests and unit tests our software is evaluated on.”

In order to believe in this process, you have to

- trust that the tests covered every single edgecase

# Formal methods: Automated reasoning, model checking, etc

Answer: We have a mathematical model of the software and used automated formal methods to check that model and specification agree.

In order to believe in this process, you have to

- trust that mathematical model matches the implementation
- trust that the automated method chosen is correct in itself

In practice, formal methods often require changing programs to fall into the scope of the methods. Big time overhead.

# Caveat: Better approaches only work in better programming languages

In order to write a formal specification, you need a programming language with formally determinable behaviour.

Out: Python, Ruby, Java, every ordinary web language like Javascript, basically every blockchain language as of today

Borderline: C, C++, new web languages like TypeScript

Good: Rust, functional programming languages like OCaml, Haskell, Standard ML or programming languages specifically developed for the task at hand

# Excursion into theory: Undecidability

The founding moment of computer science:

**The are yes-no-questions no computer can answer (Church, Turing, Post, 1930s)**

Example for such a question:

**Given a program and a specification, does the program fulfil the specification? (Rice, 1950s)**

# Undecidability in practice

Automated checking of a specification might

- take very long
- crash
- will run forever and never actually give an answer

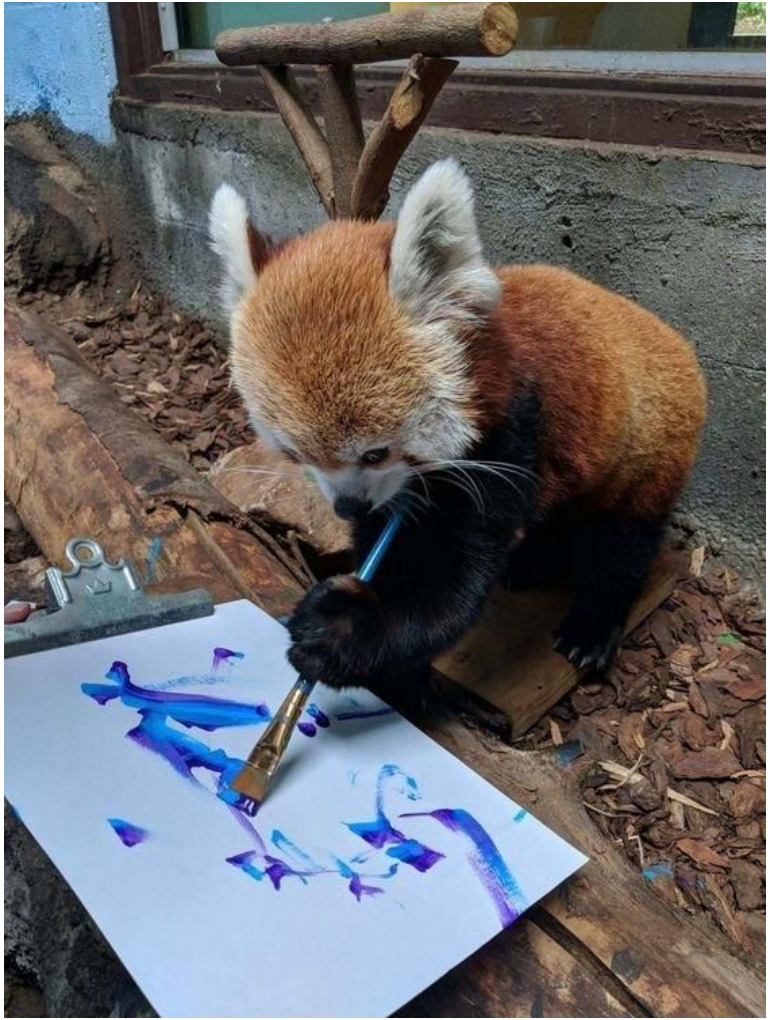
How to know whether it is taking very long or run forever?

Yet another problem unanswerable by a computer…

# Mathematical proofs about programs

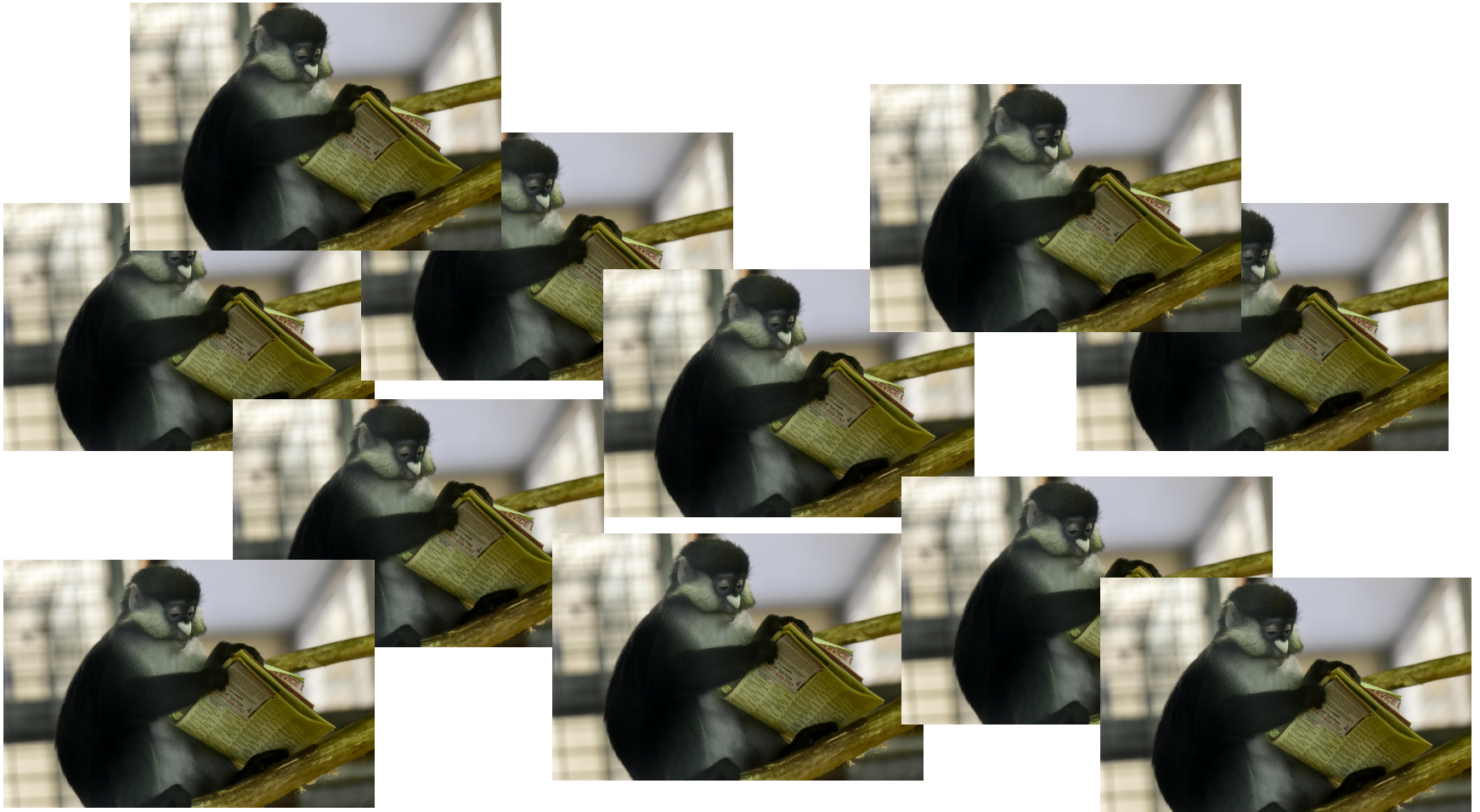

©Jan Piecha/Comedywildlifephoto.com


©Sarosh Lodhi/Comedywildlifephoto.com



by angela n., https://www.flickr.com/photos/aon/26783502295/in/photostream/, CC-BY-2.0

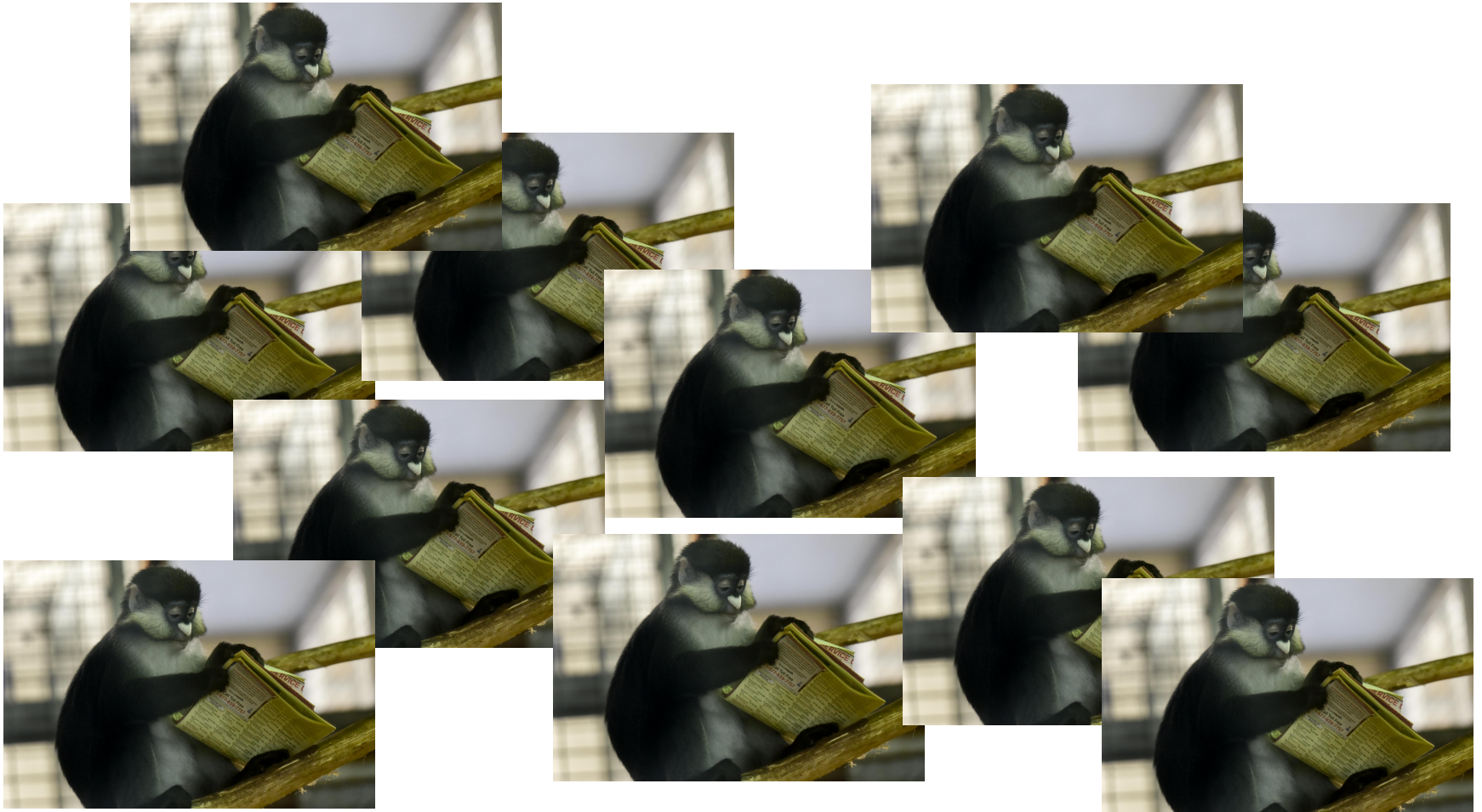

©Sarosh Lodhi/Comedywildlifephoto.com



by angela n., https://www.flickr.com/photos/aon/26783502295/in/photostream/, CC-BY-2.0


©Jan Piecha/Comedywildlifephoto.com


©Sarosh Lodhi/Comedywildlifephoto.com



by angela n., https://www.flickr.com/photos/aon/26783502295/in/photostream/, CC-BY-2.0


Inria

# The gold standard: Machine-checked correctness proofs

- Formalise semantics of programming language mathematically
- Formalise properties of the program as logical formulas
- Prove that the program fulfils the property:
  - the proof assistant checks every single argument in the proof
  - it helps keeping track of details (there are many details)
  - it can help in closing simple proof goals
  - it can help in searching for existing results

# Making assumptions precise: Trusted computing base

In any case, to trust that a program is correct you have to trust

- The compiler used to translate the program from a programming language to machine-readable code
- The runtime used to interpret this machine-readable code
- The processor used to run the machine-readable code on

If any of these is faulty, the program will be faulty.

- If automated tools are used, the same points apply.

# Machine-checked programs

You already have a program you want to prove correct?

Make programming language semantics formal, write program, prove properties

You need to write a correct program from scratch?

Use programming language built-in to proof assistant, prove properties, then use automatic extraction to industrial programming language

TCB: The proof assistant, the extraction process, and, as usual: the compiler, the runtime, the processor

# The gold gold standard: Machine-checked proof assistants

A proof assistant is a program. Thus, we can use the same methods we use to verify programs to verify the proof assistant itself.

# The MetaCoq project

- a formalisation of Coq in Coq
- machine-checked theorems of the meta theory of Coq:
  - confluence (different reduction paths converge again)
  - validity (if t has type T, then T has a type)
  - subject reduction (if t has type T and steps to term t', then t' has type T)
  - weak call-by-value standardisation (if t is of type $\mathbb{N}$ and reduces to a value, then this value can be found with weak call-by-value evaluation)
- machine-checked programs regarding Coq:
  - a correct and complete type checker
  - an erasure procedure into an untyped version of Coq, removing proofs
- Vision: a fast kernel for daily use, a verified kernel for monthly use
- Future work: eta, modules, template polymorphism

# Team effort

MetaCoq is developed by (left to right) Abhishek Anand, Danil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Meven Lennon-Bertrand, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.

# Three kinds of truth

- It is certain that a statement is true, but we have no proof
- There is a proof that a statement is true, but the proof is very complicated
- The proof can be machine-checked

We can estimate mathematical hardness well (intuition built over years)

Estimating how hard machine-checking it not so easy

Whether a proof takes a week, month or year is often hard to assess. Incremental, bottom up approach necessary

# About auxiliary results

- Do we have the auxiliary results that are just cited in the paper ready?
- If no, can we get them?
- If yes, are they in the right form? How close?
- If maybe, how can we find out how close we are?

Approach 1: First finish all auxiliary results, then work on the main theorem.

Approach 2: Just assume all auxiliary results, finish the main theorem first.

Approach in between: A bit here, a bit there.

(None of the approaches work. We need to organise our >150k codebase better, Coq needs a better Search function)

# Verified extraction

joint work with Pierre Giraud, Matthieu Sozeau, Pierre-Marie Pédrot and Nicolas Tabareau

We can verify the extraction process as well, after all it is just a program.

Different design goals:

- we want correct machine-code which can be run
- we want a correct program which can be called as subroutine from another
- we want a correct program which calls trusted subroutines
- we want a correct and human-readable program

# Intermediate language

- first-order inductives
- weak call-by-value evaluation vs reduction
- constructors as functions vs constructors as blocks
- structural fixpoints vs true fixpoints
- case representation and lets
- singleton cases
- parameter stripping
- induction principles and views
- performance (environments)

# Our concrete plan

**Theorem:** Whenever a Coq term t is well-formed, of type $\mathbb{N}$, and evaluates to a natural number n, then the translation of t into the untyped intermediate language Malfunction of the OCaml compiler evaluates to number n as well.

- we use “untyped Coq” as intermediate language
- we have formalised Malfunction
- writing an extraction function from untyped Coq to Malfunction is easy
- verifying it should be easy

# What is a well-formed term?

We only know how to prove extraction correct for terms with

- eta-expanded constructors:
  - map S l
  - map (fun x : nat => S x) l
- eta-expanded fixpoints:
  - fix f (t : tree) := match t with T (l : list tree) => sum (map f l) end
  - fix f (t : tree) := match t with T (l : list tree) => sum (map (fun x => f x) l) end
- no co-fixpoints

Every term can be converted into a well-formed term, but you have to trust this conversion process.

# Take home messages

- There are different ways how to ensure that a program is correct
  - Trustworthy ways do not work in every programming language
  - Correctness without formal methods is a debatable notion
  - Mathematical correctness proofs on paper increase trustworthiness
  - Machine-checked correctness proofs, developed in cooperation between computer
- Trade-offs are involved: Higher correctness means more time investment
- Bringing together theory and reality benefits both
- We are working on the gold gold standard of correctness:

  Programs verified in a proof assistant, which is itself verified

# Outline

1. How to ensure the correctness of programs
2. Machine-checked proofs in proof assistants
3. Machine-checked proofs about proof assistants
4. Machine-checked proofs for programs

# Disclaimer

- I consider myself a theoretical computer scientist
- This means that I try to abstract away from concrete problems as much as possible to see the bigger picture
- This talk is an example: It has nothing to do with algorithmic law, but making conceptual rather than complete points, it might be very relevant to algorithmic law
- Also, to contribute to the academic diversity of this conference, I made slides that look like theoretical computer science slides :)