# On Embedding Code Extracted From Coq Formalisations into DAWs

## deRSE25

Mario Frank

University of Potsdam
Institute of Computer Science

Karlsruhe, 26 Feb 2025

# Outline

# Motivation

In Data Analysis Workflows (DAWs), operations on data are usually implemented

- as “handwritten” code
- in languages like Python, R and C/C++
- and usually reuse specialised libraries.

But trustworthiness of the DAW results depends on the correct implementation

- of the code
- and the libraries.

# A Motivating Example (1)

A DAW analysies remote sensing data in astrophysics

- the data is used for computing the material composition of a planet
- each element is represented by an integral number (0-255)
- the “visible” side of the planet is represented as matrix of integral numbers
- the workflow includes visualisation of found elements by distinct colours
- the composition is visualised as a matrix of RGB values

# A Motivating Example (1)

A DAW analysies remote sensing data in astrophysics

- the data is used for computing the material composition of a planet
- each element is represented by an integral number (0-255)
- the “visible” side of the planet is represented as matrix of integral numbers
- the workflow includes visualisation of found elements by distinct colours
- the composition is visualised as a matrix of RGB values

# A Motivating Example (1)

A DAW analysies remote sensing data in astrophysics

- the data is used for computing the material composition of a planet
- each element is represented by an integral number (0-255)
- the “visible” side of the planet is represented as matrix of integral numbers
- the workflow includes visualisation of found elements by distinct colours
- the composition is visualised as a matrix of RGB values

# A Motivating Example (1)

A DAW analysies remote sensing data in astrophysics

- the data is used for computing the material composition of a planet
- each element is represented by an integral number (0-255)
- the “visible” side of the planet is represented as matrix of integral numbers
- the workflow includes visualisation of found elements by distinct colours
- the composition is visualised as a matrix of RGB values

# A Motivating Example (1)

A DAW analysies remote sensing data in astrophysics

- the data is used for computing the material composition of a planet
- each element is represented by an integral number (0-255)
- the “visible” side of the planet is represented as matrix of integral numbers
- the workflow includes visualisation of found elements by distinct colours
- the composition is visualised as a matrix of RGB values

# A Motivating Example (2)

To visualise the found elements in a publication or internal documents,

- the matrix of integers is transformed into a matrix of RGB values
- by applying a function `int id_to_rgb(int e)` on each element
- and then storing the result as a bitmap

If `id_to_rgb` is not **correctly** implemented, results can be misinterpreted.

# What is Correctness?

An algorithm is correct, if it satisfies a given specification that usually defines

- input and output types
- input and output constraints (e.g. restrictions on values)
- the (mathematical) function computed with inputs

For example, if

- `e` is a non-negative integer smaller than 256
- `id_to_rgb(e)` must be non-negative, too.

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

# How to Ensure Correctness?

Approaches can be

1. Using Assertions (in code)
2. Testing (unit-tests)
3. Model Checking
4. Formal Verification in Proof Assistants
5. Correct-by-Construction Synthesis

However,

- 1-2 show only the **absence** of **specific** errors
- and only 3-5 can guarantee correctness.

# Proof Assistants

Proof Assistants can be used to

1. define and verify mathematical propositions and laws
2. encode scientific theories (like climate models)
3. define and verify properties of algorithms

And the most prominent are

- Rocq [1] (aka Coq)
- Isabelle/HOL [2]
- Lean [3]

# Synthesis and Extraction

Some Proof Assistants are capable of

1. constructing a functional model from a proof (synthesis)
2. extracting compilable/runnable code from a functional model

In the best case, extraction is verified as for

- Isabelle/HOL (extraction to CakeML [4])
- Rocq (many target languages)

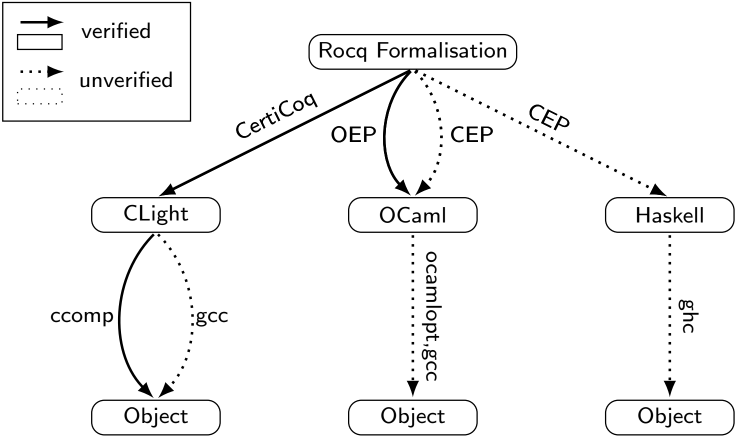
# Code Extraction in Rocq (1)

Rocq supports extraction to source code with the

1. Coq Extraction Plugin (CEP) [5]: OCaml, Haskell, Scheme
2. OCaml Extraction Plugin (OEP) [6]: OCaml
3. CertiCoq [8]: Clight [7]
4. and more (WebAssembly, Rust)

And to wrap up in a graphic

# Code Extraction in Rocq (2)



# Restrictions of Extraction

Usually, the extracted code does not contain

- a “main” function - since it is not pure
- declarations for Foreign Function Interfaces (FFIs)

So these have to be defined manually.

# Reuse Options

Generally, there are two ways of reusing extracted code in a DAW

1. as a standalone DAW step (tool)
   - Requires CLI skeleton
   - Requires data input/output functionality

2. as a part of a DAW step (integration)
   - homogenous (same programming language)
   - heterogenous (e.g. OCaml in C++)

→ Tool extraction and homogenous integration are rather straight-forward

# Why Heterogenous Integration?

OCaml (and Haskell) have advantages

- data structures can exceed usual limitations (e.g. 64 bit integers)
  $\rightarrow$ higher computation precision possible
- purely functional algorithms are SIMD
  $\rightarrow$ can be parallelised easily (e.g. OCaml 5)
- functional languages are strongly type safe
  $\rightarrow$ less error-prone compared to Python, for example
- both can be compiled to object code

# Heterogenous Integration with FFIs

OCaml has FFIs to C/C++ [10], but

- C-callable OCaml functions have to be exposed as callbacks
- OCaml-callable C functions have to be defined as externals
- a type conversion has to be done

Usually, all this is done manually!

# More Technical

When having

- a C function `int my_fun (int a, int b)` and
- an extracted OCaml function `my_ocaml_fun : int -> int -> int`,

we have to manually define the

1. external declaration in OCaml
   `external my_fun : int -> int -> int`
2. callback declaration in OCaml
   `let _ = Callback.register "my_callback" my_ocaml_fun`
3. implementation of `my_fun` and call to `my_callback` in C.

# The Problem

Changes on the formalisation may require updating externals/callbacks!

Otherwise, runtime errors may occur when

- calling C externals (typing errors)
- calling non-existing callbacks (if OCaml side name changes)

# The Solution

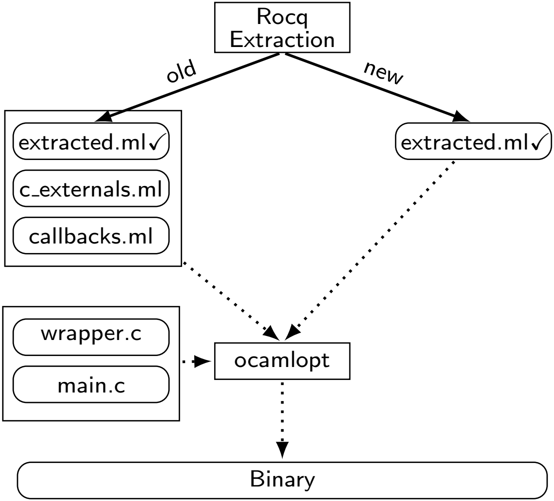
In Rocq,

- the target OCaml type after extraction is defined
  $\rightarrow$ So the typing of the external definition is already available
- the names of potential callbacks are present
  $\rightarrow$ So the callback registration info is already available

Thus, we extended [12] the CEP with commands [13]

- `Extract Foreign Constant qualid` that generate a OCaml-side type save externals declaration for the `qualid` function
- `Extract Callback qualid` to automatically generate a callback registration for the `qualid` function

# Extraction Comparison



# The Current State

The changes in the extraction plugin of Rocq

- increase type safety when integrating extracted code into C/C++ programs
- make exposing foreign functions between C/C++ and OCaml easier
- and can in principle also be used by the verified OCaml extraction

# The Limitations

But there are open topics:

- the extensions support only FFIs to C/C++ but not Python
  $\rightarrow$ potential future work by leveraging pyml [11].
- the C side type conversions cannot be extracted by the CEP
  $\rightarrow$ potential future work as separate Rocq plugin
- the C side calls to OCaml have to be implemented manually
  $\rightarrow$ potential future work as separate Rocq plugin

In that sense:

# Better Software $\rightarrow$ Better Research

# References


1 Y. Bertot & P. Castéran (2004): “Interactive Theorem Proving and Program Development - Coq’Art: The Calculus of Inductive Constructions”. https://doi.org/10.1007/978-3-662-07964-5.

2 T. Nipkow, L. C. Paulson & M. Wenzel (2002): “Isabelle/HOL - A Proof Assistant for Higher-Order Logic”. Lecture Notes in Computer Science, Springer Berlin, Heidelberg. https://doi.org/10.1007/3-540-45949-9.

3 The Lean team (2025): “Programming Language and Theorem Prover”. https://lean-lang.org/

4 L. Hupel & T. Nipkow (2018): “A Verified Compiler from Isabelle/HOL to CakeML”. In Amal Ahmed, editor: European Symposium on Programming (ESOP), Lecture Notes in Computer Science 10801, Springer, pp. 999–1026. https://doi.org/10.1007/978-3-319-89884-1_35.

5 P. Letouzey (2003): “A New Extraction for Coq”, In Herma Geuvers & Freek Wiedijk, editors: Types for Proofs and Programs, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–219, https://doi.org/10.1007/3-540-39185-1_12.

6 Y. Forster, M. Sozeau & N. Tabareau (2023): “Verified Extraction from Coq to OCaml”, https://inria.hal.science/hal-04329663

7 S. Blazy & X. Leroy (2009): “Mechanized Semantics for the Clight Subset of the C Language”, Journal of Automated Reasoning 43(3), pp. 263–288, https://doi.org/10.1007/s10817-009-9148-3.

8 A. Anand et al. (2017): “CertiCoq : A verified compiler for Coq”, In CoqPL’17: The Third International Workshop on Coq for Programming Languages.

9 The Rocq developers (2024): “Program extraction”, https://coq.inria.fr/doc/v8.20/refman/addendum/extraction.html.

10 INRIA (2025): “Interfacing C with OCaml”, https://ocaml.org/manual/5.3/intfc.html

11 The `pyml` developers (2023): ”OCaml bindings for Python“, urlhttps://github.com/ocamllibs/pyml.

12 M. Frank (2024): ”Extend the Extraction Plugin to synthesise OCaml external and callback definitions for interfacing C/C++ https://github.com/coq/coq/pull/18270/.

13 INRIA (2024): https://coq.inria.fr/doc/v8.20/refman/addendum/extraction.html

# Acknowledgements

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through the VerSeCloud research project under the grant number 16KIS1358