

Rocq's Conversion and Reduction Engines

Guillaume Melquiond

April 20, 2026

The Most Interesting Rule in Rocq's Type Theory

Conversion

$$\frac{E[\Gamma] \vdash U : s \quad E[\Gamma] \vdash t : T \quad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}$$

In a nutshell

If the terms T and U have the same normal form, then any proof t of T is also a proof of U .

The Most Interesting Rule in Rocq's Type Theory

Conversion

$$\frac{E[\Gamma] \vdash U : s \quad E[\Gamma] \vdash t : T \quad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}$$

In a nutshell

If the terms T and U have the same normal form, then any proof t of T is also a proof of U .

Example (on natural numbers)

A proof of $2 = 2$ is a proof of $1 + 1 = 2$.

A proof of $x = x$ is a proof of $0 + x = x$.

A proof of $x = x$ is not a proof of $0 + x = x + 0$ (in Rocq).

Outline

- 1 Introduction
 - Conversion
 - Computational reflection
- 2 Lazy conversion
- 3 Primitive types and operations
- 4 Bytecode and native reduction

Proofs by Computational Reflection

How to prove some property Q using computations?

- 1 Define an inductive type I , a function $f : I \rightarrow \mathbb{B}$, and a predicate $P : I \rightarrow \text{Prop}$.
- 2 Prove $\forall x : I, f(x) = \text{true} \Rightarrow P(x)$ once and for all.
- 3 Find y such that Q is convertible to $P(y)$.
- 4 Use `(eq_refl true)` as a proof of $f(y) = \text{true}$.

Proofs by Computational Reflection

How to prove some property Q using computations?

- 1 Define an inductive type I , a function $f : I \rightarrow \mathbb{B}$, and a predicate $P : I \rightarrow \text{Prop}$.
- 2 Prove $\forall x : I, f(x) = \text{true} \Rightarrow P(x)$ once and for all.
- 3 Find y such that Q is convertible to $P(y)$.
- 4 Use `(eq_refl true)` as a proof of $f(y) = \text{true}$.

Challenges for the kernel

- Check **smartly** $Q \equiv P(y)$.
- Check **efficiently** $f(y) \equiv \text{true}$.

Outline

- 1 Introduction
- 2 Lazy conversion
 - Lazy conversion
 - Tricks, part 1
- 3 Primitive types and operations
- 4 Bytecode and native reduction

Lazy Conversion

Example (on natural numbers)

Terms `(fact 15)` and `(15 * fact 14)` are convertible.
But computing their normal form is impossible.

Lazy Conversion

Example (on natural numbers)

Terms $(\text{fact } 15)$ and $(15 * \text{fact } 14)$ are convertible.
But computing their normal form is impossible.

Heuristics

- 1 If T and U have the same shape, compare component-wise, from right to left. Stop if all the subterms are convertible component-wise.
- 2 If T is $f(\vec{x})$ and U is $g(\vec{y})$, ask an oracle which of f or g should be unfolded first.
- 3 Go back to step 1.

Lazy Conversion

Example (on natural numbers)

Terms $(\text{fact } 15)$ and $(15 * \text{fact } 14)$ are convertible.
But computing their normal form is impossible.

Heuristics

- 1 If T and U have the same shape, compare component-wise, from right to left. Stop if all the subterms are convertible component-wise.
- 2 If T is $f(\vec{x})$ and U is $g(\vec{y})$, ask an oracle which of f or g should be unfolded first.
- 3 Go back to step 1.

Controlling the oracle

Strategy, from $-\infty$ (expand) to 0 (Transparent, by default) to $+\infty$ (Opaque).

Tricks, Part 1

Discarding non-informative content

Use decidable properties to discard opaque terms by computation.

Evaluating functions defined by well-funded induction

Pump extra accessibility constructors before computation.

Outline

- 1 Introduction
- 2 Lazy conversion
- 3 Primitive types and operations
 - 63-bit integers
 - 64-bit floating-point numbers
 - Persistent arrays
 - Strings
- 4 Bytecode and native reduction

Primitive Types

- More compact representation of some values.
- Faster manipulation of some values.
- Conservative extension of Rocq's theory.

63-bit Integers, Part 1

Representation

OCaml native integers on 64-bit architecture.

Arithmetic operations

- Modulo operations: addition, subtraction, multiplication.
- Signed and unsigned division and remainder.
- Signed and unsigned comparisons.

Bit-level manipulations

- Bitwise and, or, xor.
- Signed and unsigned shifts.
- Count leading and trailing zeros.

63-bit Integers, Part 2

Long arithmetic operations

- Addition, subtraction $63 + 63 + 1 \rightarrow (63, 1)$.
- Multiplication $63 \times 63 \rightarrow (63, 63)$. Division $(63, 63) \div 63 \rightarrow 63$.
- Shifts $(63, 63) \ll 63 \rightarrow 63$.

Library Bignums

- Integers represented as balanced binary trees with int63 leaves.
- Sub-quadratic multiplication, division, and square root.
- GCD and rational numbers.

64-bit Floating-Point Numbers, Part 1

Representation

OCaml floating-point numbers but without NaN payload.

IEEE-754-like arithmetic

- Addition, multiplication, division, square root.
- Comparisons.
- Exponent and significand manipulation.

Library Flocq

Proper “as if” specification using real arithmetic.

64-bit Floating-Point Numbers, Part 2

Adhoc operations

- Predecessor and successor.

The Interval library

- Intervals with floating-point bounds.
- Polynomial approximations.
- Definite integrals.

Persistent Arrays

Representation

- OCaml arrays with a list of modifications.
- $O(1)$ read/write to the latest version of an array.

Operations

- Get and set with `int63` indices.
- Copy.

Strings

Representation

OCaml strings.

Operations

- Get with `int63` indices and characters.
- Substring, concatenation, and length.
- Lexicographic comparison.

Outline

- 1 Introduction
- 2 Lazy conversion
- 3 Primitive types and operations
- 4 Bytecode and native reduction
 - Normalization by evaluation
 - Translation to OCaml
 - Toplevel constants
 - Tricks, part 2

The Most Interesting Rule in Rocq's Type Theory, Again

Conversion by normalization

$$\frac{E[\Gamma] \vdash U : s \quad E[\Gamma] \vdash t : T \quad E[\Gamma] \vdash T' \leq_{\beta\delta\iota\zeta\eta} U' \quad E[\Gamma] \vdash T \rightsquigarrow_{\beta\delta\iota\zeta} T' \quad E[\Gamma] \vdash U \rightsquigarrow_{\beta\delta\iota\zeta} U'}{E[\Gamma] \vdash (t <: U) : U}$$

Usually, t is a term with a trivial type, which is already in normal form, e.g., `eq_refl true : true = true`.

Evaluation

Values

- Abstraction $\text{fun } x \Rightarrow e$.
- Opaque symbol (e.g., constructor) applied to values.
- `match` or `fix` applied to a non-constructor value.

Evaluation

Values

- Abstraction $\text{fun } x \Rightarrow e$.
- Opaque symbol (e.g., constructor) applied to values.
- `match` or `fix` applied to a non-constructor value.

Reduction rules

- $f(e_1, \dots, e_n)$: evaluate f , e_1, \dots, e_n , and then β -reduce if possible.
- $\text{match}(e, \{C_i \Rightarrow b_i\})$: evaluate e , and then select a branch b_i if possible.

Evaluation

Values

- Abstraction $\text{fun } x \Rightarrow e$.
- Opaque symbol (e.g., constructor) applied to values.
- `match` or `fix` applied to a non-constructor value.

Reduction rules

- $f(e_1, \dots, e_n)$: evaluate f , e_1, \dots, e_n , and then β -reduce if possible.
- $\text{match}(e, \{C_i \Rightarrow b_i\})$: evaluate e , and then select a branch b_i if possible.

Modulo “if possible”, those are the evaluation rules of OCaml.

Normalization by Evaluation

Computing the normal form of e

- 1 Evaluate e into a value v .
- 2 Extract all the abstractions $(\lambda x. e_i)$ from v .
- 3 Recursively go to step 1 to normalize $e'_i = (\lambda x. e_i)y$ with y a fresh symbol.
- 4 Abstract all the e'_i with respect to y and plug them into v . Return v .

Normalization by Evaluation

Computing the normal form of e

- 1 Evaluate e into a value v .
- 2 Extract all the abstractions $(\lambda x. e_i)$ from v .
- 3 Recursively go to step 1 to normalize $e'_i = (\lambda x. e_i)y$ with y a fresh symbol.
- 4 Abstract all the e'_i with respect to y and plug them into v . Return v .

Example (Normalization of $(1 + 1, (\lambda x. 0 + x))$)

- 1 $(1 + 1, (\lambda x. 0 + x)) \rightsquigarrow (2, (\lambda x. 0 + x))$.
- 2 $(\lambda x. 0 + x)y \rightsquigarrow y$.
- 3 Normal form is $(2, (\lambda y. y))$.

Bytecode and Native Reduction

Bytecode

- Compiled (resp. executed) by a modified OCaml bc compiler (resp. interpreter).
- Cast: $t <: U$.
- Tactics: `vm_compute` and `vm_cast_no_check`.

Bytecode and Native Reduction

Bytecode

- Compiled (resp. executed) by a modified OCaml bc compiler (resp. interpreter).
- Cast: $t <: U$.
- Tactics: `vm_compute` and `vm_cast_no_check`.

Native

- Compiled to machine code by the OCaml native compiler.
- Cast: $t <<: U$.
- Tactics: `native_compute` and `native_cast_no_check`.

Translation to OCaml

A mostly straightforward translation

- Abstractions (resp. let-ins) are translated to OCaml functions (resp. let-ins).

Translation to OCaml

A mostly straightforward translation

- Abstractions (resp. let-ins) are translated to OCaml functions (resp. let-ins).
- Constructors of inductive types are translated to OCaml constructors, but parameters of inductive types are dropped (hence not normalized).

Translation to OCaml

A mostly straightforward translation

- Abstractions (resp. let-ins) are translated to OCaml functions (resp. let-ins).
- Constructors of inductive types are translated to OCaml constructors, but parameters of inductive types are dropped (hence not normalized).
- Opaque symbols are compiled to a special function that gobbles its arguments.

Translation to OCaml

A mostly straightforward translation

- Abstractions (resp. let-ins) are translated to OCaml functions (resp. let-ins).
- Constructors of inductive types are translated to OCaml constructors, but parameters of inductive types are dropped (hence not normalized).
- Opaque symbols are compiled to a special function that gobbles its arguments.
- Pattern-matching is translated to OCaml pattern-matching, but with an extra branch when the matched value is an accumulator.

Translation to OCaml

A mostly straightforward translation

- Abstractions (resp. let-ins) are translated to OCaml functions (resp. let-ins).
- Constructors of inductive types are translated to OCaml constructors, but parameters of inductive types are dropped (hence not normalized).
- Opaque symbols are compiled to a special function that gobbles its arguments.
- Pattern-matching is translated to OCaml pattern-matching, but with an extra branch when the matched value is an accumulator.
- Recursive functions are translated as OCaml functions, but with a check that their structural argument is a constructor.

Compilation and Linking of Toplevel Constants

Bytecode

- Constants are compiled just after their definition.
- Compiled constants are stored in the .vo file.
- Linking happens at `vm_compute` time.
- Constants are lazily evaluated and cached during linking.

Compilation and Linking of Toplevel Constants

Bytecode

- Constants are compiled just after their definition.
- Compiled constants are stored in the `.vo` file.
- Linking happens at `vm_compute` time.
- Constants are lazily evaluated and cached during linking.

Native

- Constants are precompiled at the end of `coqc` (or when calling `coqnative`).
- Compiled constants are stored in a `.cmxs` file.
- Non-precompiled constants are compiled at the time of `native_compute`.
- Linking happens at `native_compute` time (only once).
- Constants are lazily evaluated and cached during execution.

Tricks, Part 2

Caching costly computations

- `abstract + vm_cast_no_check`.
- Toplevel constants.
- Coinductive values.