Verifying dependent type-checkers

WITS 2026

Meven Lennon-Bertrand

I've written one type-checker in my life, and for a rather simple theory...

A disclaimer

I’ve written one type-checker in my life, and for a rather simple theory…

so I’m going to do **propaganda**!

Because that type-checker was **fully verified!**

Verifying the verifier

We keep telling the world they should verify their critical code...

Verifying the verifier

We keep telling the world they should verify their critical code...
A lot of the verification ecosystem relies on proof assistant kernels/dependent type-checkers...

We keep telling the world they should verify their critical code…
A lot of the verification ecosystem relies on proof assistant kernels/dependent type-checkers…

Why don't we still have verified kernels?

We keep telling the world they should verify their critical code…
A lot of the verification ecosystem relies on proof assistant kernels/dependent type-checkers…

Why don’t we still have verified kernels?

The programs are not that complicated…

We keep telling the world they should verify their critical code...
A lot of the verification ecosystem relies on proof assistant kernels/dependent type-checkers...

Why don't we still have verified kernels?

The programs are not that complicated...

But the reasons why they work are complicated!

Logical relations

No computational content

Meta-theory

Models

Algebraic

Presentation-free

Execution/Extraction

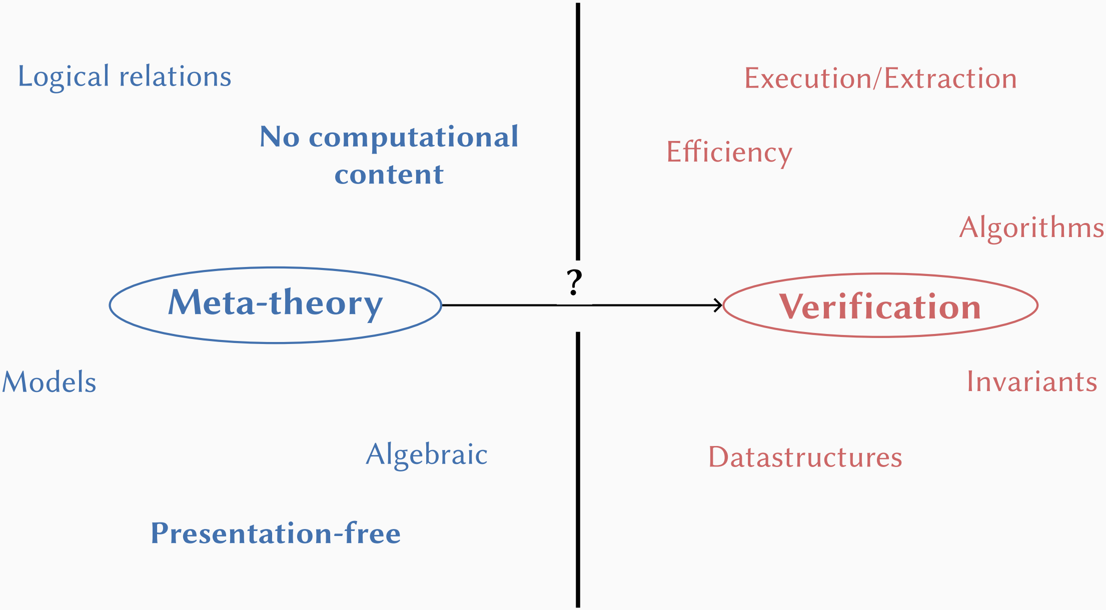
Efficiency

Algorithms

Verification

Invariants

Datastructures


Logical relations
No computational content
Execution/Extraction
Efficiency
Algorithms
Meta-theory
?
Verification
Models
Invariants
Algebraic
Datastructures
Presentation-free

All hail verification

Much focus on the **meta-theory** side

Much focus on the **meta-theory** side

But a lot of interesting questions on the **verification** side:

- What is the abstract specification of your type system?
- How do the datastructures you use relate to their abstract variants?
- What invariants do your code & datastructures rely on?
- What meta-theoretic properties are needed to verify these invariants?

Much focus on the **meta-theory** side

But a lot of interesting questions on the **verification** side:

- What is the abstract specification of your type system?
- How do the datastructures you use relate to their abstract variants?
- What invariants do your code & datastructures rely on?
- What meta-theoretic properties are needed to verify these invariants?

Particularly worth asking for **realistic implementations**

We should **verify implementations** even if **we don't do the meta-theory**

This talk

We should **verify implementations** even if **we don't do the meta-theory**

A tour of case studies in that space:

- pattern-matching in MetaRocq
  - *Correct and Complete Type Checking and Certified Erasure for Coq, in Coq* (JACM 2025)
  - *The Curious Case of Case: Correct & Efficient Representation of Case Analysis in Coq and MetaCoq* (WITS 2022)
- verifying untyped conversion
  - *Martin-Löf à la Coq* (CPP 2024)
  - *What Does It Take to Certify a Conversion Checker?* (FSCD 2025)

MetaRocq: the curious case of case

The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

A dependent type theory with

- Crazy (co-)inductive types
- Pattern-matching and fixed-points
- Fancy universes + cumulativity

The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

A dependent type theory with

- Crazy (co-)inductive types
- Pattern-matching and fixed-points
- Fancy universes + cumulativity

Rocq, in Rocq

- **Formalised meta-theory of PCUIC**
- Normalisation axiom to implement a **verified type-checker**
- **Verified extraction**
- Meta-programming

The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

A dependent type theory with

- Crazy (co-)inductive types
- Pattern-matching and fixed-points
- Fancy universes + cumulativity

Rocq, in Rocq

- **Formalised meta-theory of PCUIC**
- Normalisation axiom to implement a **verified type-checker**
- **Verified extraction**
- Meta-programming

We found a bug in Rocq!

Coq 6.1	Pattern-matching (Cornes), representation chosen for backwards compatibility
Coq 8.4	Universe polymorphism (Sozeau & Tabareau)
Coq 8.7	Cumulative inductive types, theory **for eliminators** (Sozeau & Timany)
Meanwhile	People like less and less the clunky pattern-matching representation

Coq 6.1	Pattern-matching (Cornes), representation chosen for backwards compatibility
Coq 8.4	Universe polymorphism (Sozeau & Tabareau)
Coq 8.7	Cumulative inductive types, theory **for eliminators** (Sozeau & Timany)
Meanwhile	People like less and less the clunky pattern-matching representation
Nov. ’20	We are trying to prove type-checking is complete

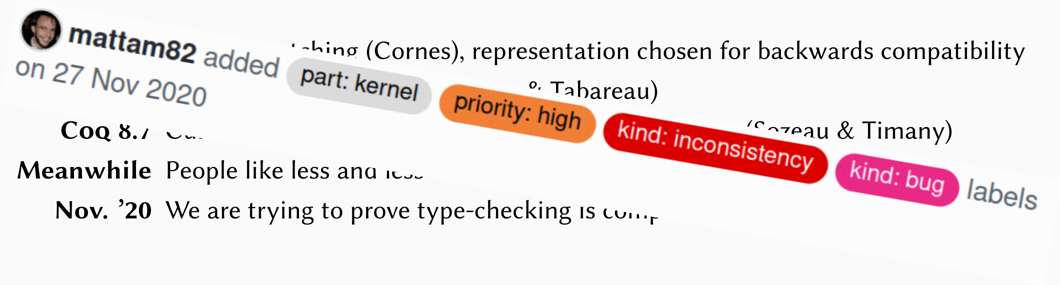
…ching (Cornes), representation chosen for backwards compatibility

… & Tabareau)

Coq 8.7 Cu… (Sozeau & Timany)

Meanwhile People like less and less…

Nov. ’20 We are trying to prove type-checking is comp…



Post-mortem of a bug

Coq 6.1	Pattern-matching (Cornes), representation chosen for backwards compatibility
Coq 8.4	Universe polymorphism (Sozeau & Tabareau)
Coq 8.7	Cumulative inductive types, theory **for eliminators** (Sozeau & Timany)
Meanwhile	People like less and less the clunky pattern-matching representation
Nov. ’20	We are trying to prove type-checking is complete
Coq 8.13	**Kernel bug**! ⟶ quick and dirty fix
Coq 8.14	Complete redesign, in parallel in Rocq and MetaRocq

Issue #13495

```
match s as x in Ind _ inds return P with
…
end
```

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13:

1. infer the type Ind $\vec{p}\,\vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}).\, \square_?$
3. check the branches
4. return $P\,\vec{i}\,\vec{s}$

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13:

1. infer the type Ind $\vec{p}\,\vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}).\ \square_{?}$
    - infer the type of P
    - check it is of the form $\Pi\,\Delta.\,\square_l$
    - check that $\Delta \equiv (\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds})$
3. check the branches
4. return $P\ \vec{i}\ \vec{s}$

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13: **Can you spot the issue?**

1. infer the type $\mathrm{Ind}\ \vec{p}\,\vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}).\ \square_?$
    - infer the type of P
    - check it is of the form $\Pi\,\Delta.\ \square_l$
    - check that $\Delta \equiv (\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds})$
3. check the branches
4. return $P\ \vec{i}\ \vec{s}$

Issue #13495

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13: **Can you spot the issue?**

1. infer the type $\mathrm{Ind}\ \vec{p}\,\vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\,\overrightarrow{inds}).\,\square_?$
   - infer the type of P
   - check it is of the form $\Pi\,\Delta.\,\square_l$
   - check that $\Delta \equiv (\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\,\overrightarrow{inds})$
3. check the branches
4. return $P\,\vec{i}\,\vec{s}$

Issue #13495

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13:

1. infer the type $\mathrm{Ind}\ \vec{p}\,\vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}).\,\square_?$
   - infer the type of P
   - check it is of the form $\Pi\,\Delta.\,\square_l$
   - check that $\Delta \succeq (\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds})$
3. check the branches
4. return $P\ \vec{i}\ \vec{s}$

Issue #13495

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13:

1. infer the type $\mathrm{Ind}\ \vec{p}\ \vec{i}$ of s
2. "check" P against $\Pi(\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}).\ \square_?$
   - infer the type of P
   - check it is of the form $\Pi\,\Delta.\ \square_l$
   - check that $\Delta \succeq (\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds})$
3. check the branches
4. return $P\ \vec{i}\ \vec{s}$

8.14 – 9.x: check that P is a type in a context extended by $\overrightarrow{inds} : \mathrm{Indices}_{\mathrm{Ind}}[\vec{p}], x : \mathrm{Ind}\ \vec{p}\ \overrightarrow{inds}$

```
match s as x in Ind _ inds return P with
…
end
```

6.1 – 8.13:

1. infer the type Ind $\vec{p}\,\vec{i}$ of s
2. “check” P against $\Pi(\overrightarrow{inds} : \text{Indices}_{\text{Ind}}[\vec{p}], x : \text{Ind}\ \vec{p}\ \overrightarrow{inds}).\ \square_?$
   - infer the type of P
   - check it is of the form $\Pi\,\Delta.\ \square_l$
   - check that $\Delta \succeq (\overrightarrow{inds} : \text{Indices}_{\text{Ind}}[\vec{p}], x : \text{Ind}\ \vec{p}\ \overrightarrow{inds})$
3. check the branches
4. return $P\ \vec{i}\ \vec{s}$

8.14 – 9.x: check that P is a type in a context extended by $\overrightarrow{inds} : \text{Indices}_{\text{Ind}}[\vec{p}], x : \text{Ind}\ \vec{p}\ \overrightarrow{inds}$

Subtle **implementation mis-design**, on a **combination of advanced features**
Never described on paper or formalised

What do we want to verify?
What does it take?

Declarative specification

Arbitrarily mixing:

- Refl./Sym./Trans.
- Congruences
- Computation (β)
- Extensionality (η)

Typed!

Declarative specification

Arbitrarily mixing:

- Refl./Sym./Trans.
- Congruences
- Computation (β)
- Extensionality (η)

Typed!

Type-directed algo.

Alternate

1. β-reduction to whnf
2. **Type**-directed η
3. **Head** congruences

Declarative specification

Arbitrarily mixing:

- Refl./Sym./Trans.
- Congruences
- Computation (β)
- Extensionality (η)

Typed!

Type-directed algo.

Alternate

1. β-reduction to whnf
2. **Type**-directed η
3. Head congruences

“Untyped” algo.

Alternate

1. β-reduction to whnf
2. **Term**-directed η
3. Head congruences

Declarative specification

Arbitrarily mixing:

- Refl./Sym./Trans.
- Congruences
- Computation (β)
- Extensionality (η)

Typed!

Type-directed algo.

Alternate

1. β-reduction to whnf
2. Type-directed η
3. Head congruences

+ closer to specification
+ supports fancier rules
- slower

“Untyped” algo.

Alternate

1. β-reduction to whnf
2. Term-directed η
3. Head congruences

+ faster
+ simpler
- further from spec.

What to verify?

$$P : D \to \mathbb{P} \quad \overset{?}{\Leftrightarrow} \quad p : D \to \mathbb{B}$$

What to verify?

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \to \mathbb{B}$$

0. decidability: $(p\ d = \text{true}) \lor (p\ d = \text{false})$

What to verify?

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \to \mathbb{B}$$

0. decidability: $(p\ d = \text{true}) \vee (p\ d = \text{false})$

1. soundness: $p\ d = \text{true} \Rightarrow P\ d$

2. completeness: $P\ d \Rightarrow p\ d = \text{true}$

3. profit!

What to verify?

$$P : D \to \mathbb{P} \quad \overset{?}{\Leftrightarrow} \quad p : D \to \mathbb{B}$$

0. decidability: $(p\ d = \text{true}) \vee (p\ d = \text{false})$

1. soundness: $p\ d = \text{true} \Rightarrow P\ d$

2. completeness: $P\ d \Rightarrow p\ d = \text{true}$

3. **profit?**

What to verify?

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \rightharpoonup \mathbb{B}$$

0. **decidability**: $(p\ d = \text{true}) \lor (p\ d = \text{false})$
   How do we know the type-checker terminates?
1. soundness: $p\ d = \text{true} \Rightarrow P\ d$
2. completeness: $P\ d \Rightarrow p\ d = \text{true}$
3. profit?

What to verify?

$$P : D \to \mathbb{P} \quad \overset{?}{\Leftrightarrow} \quad p : D \to \mathbb{B}$$

0. **decidability**: $(p\ d = \text{true}) \vee (p\ d = \text{false})$
   How do we know the type-checker terminates?
1. **soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker
2. completeness: $P\ d \Rightarrow p\ d = \text{true}$

3. profit?

What to verify?

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \rightharpoonup \mathbb{B}$$

0. **decidability**: $(p\ d = \text{true}) \vee (p\ d = \text{false})$
   How do we know the type-checker terminates?
1. **soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker
2. **completeness**: $P\ d \Rightarrow p\ d = \text{true}$
   reflexivity ⇒ termination
3. profit?

What to verify?

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \rightharpoonup \mathbb{B}$$

1. **positive soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker

$$P : D \to \mathbb{P} \qquad \overset{?}{\Leftrightarrow} \qquad p : D \rightharpoonup \mathbb{B}$$

1. **positive soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker
2. **negative soundness**: $p\ d = \text{false} \Rightarrow \neg\,(P\ d)$
   Look (harder) at the trace of the type-checker

What to verify?

$$P : D \to \mathbb{P} \qquad \stackrel{?}{\Leftrightarrow} \qquad p : D \rightharpoonup \mathbb{B}$$

1. **positive soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker
2. **negative soundness**: $p\ d = \text{false} \Rightarrow \neg\,(P\ d)$
   Look (harder) at the trace of the type-checker
3. **termination**: $(p\ d = \text{true}) \vee (p\ d = \text{false})$
   Still hard, of course...

What to verify?

$$P : D \to \mathbb{P} \quad \overset{?}{\Leftrightarrow} \quad p : D \rightharpoonup \mathbb{B}$$

1. **positive soundness**: $p\ d = \text{true} \Rightarrow P\ d$
   Look at the trace of the type-checker
2. **negative soundness**: $p\ d = \text{false} \Rightarrow \neg\,(P\ d)$
   Look (harder) at the trace of the type-checker
3. **termination**: $(p\ d = \text{true}) \vee (p\ d = \text{false})$
   Still hard, of course...

A much better plan

What meta-theory do we need?

	Positive soundness	Negative soundness (typed conversion)	Negative soundness (untyped conversion)	Termination
Injectivity of type constructors	×	×	×	×
Term-level injectivities		×	×	
Normalisation				×

What meta-theory do we need?

	Positive soundness	Negative soundness (typed conversion)	Negative soundness (untyped conversion)	Termination
Injectivity of type constructors	×	×	×	×
Term-level injectivities		×	×	
Normalisation				×

Injectivities are the important properties

What meta-theory do we need?

	Positive soundness	Negative soundness (typed conversion)	Negative soundness (untyped conversion)	Termination
Injectivity of type constructors	×	×	×	×
Term-level injectivities		×	×	
Normalisation				×

Injectivities are the important properties

To verify "untyped" conversion, you still need typing invariants

Injectivity of type constructors

If $\Gamma \vdash T \equiv T'$ and T, T' are weak-head normal form, then:

- $T = \mathbb{N} = T'$
- or $T = \Pi\, x\colon A.B$, $T' = \Pi\, x\colon A'.B'$, with $\Gamma \vdash A' \equiv A$ and $\Gamma, x\colon A' \vdash B \equiv B'$
- or ...
- or T, T' are both neutrals, and $\Gamma \vdash T \equiv T' : \square$

Any non-diagonal case is impossible (*no-confusion*).

Injectivity of type constructors

Injectivity and no-confusion at $\mathbb{N}$

If $\Gamma \vdash n \equiv n' : \mathbb{N}$ and n, n' are weak-head normal forms, then:

- $n = 0 = n'$
- or $n = \mathrm{S}(t)$, $n' = \mathrm{S}(t')$, with $\Gamma \vdash t \equiv t' : \mathbb{N}$
- or n, n' are both neutrals.

Injectivity of type constructors

Injectivity and no-confusion at $\mathbb{N}$

…

Normalisation

Inductive predicate: "iterated weak-head normalisation and η-expansion terminates".

Understanding "untyped" conversion-checking

What meta-theory do we need?

	Positive soundness	Negative soundness (typed conversion)	Negative soundness (untyped conversion)	Termination
Injectivity of type constructors	×	×	×	×
Term-level injectivities		×	×	
Normalisation				×

Injectivities are the important properties

To verify "untyped" conversion, you still need typing invariants

There is a catch on neutrals!

Neutrals

Traditional PL: evaluation of closed, first-order values
Dependent types: conversion under binders, must consider **open terms**

Neutrals

Traditional PL: evaluation of closed, first-order values
Dependent types: conversion under binders, must consider **open terms**

A neutral is:

- a variable
- or an elimination, stuck on a neutral

$x : \mathbb{N} \to \mathbb{N} \times \mathbb{N}, \ldots \vdash \mathrm{rec}_{\mathbb{N}}(\pi_1(x\ 7), P, b_0, b_S) : P$

Traditional PL: evaluation of closed, first-order values
Dependent types: conversion under binders, must consider **open terms**

A neutral is:

- a variable
- or an elimination, stuck on a neutral

$x : \mathbb{N} \to \mathbb{N} \times \mathbb{N}, \ldots \vdash \mathrm{rec}_{\mathbb{N}}(\pi_1(x\ 7), P, b_0, b_S) : P$

How does one compare neutrals? Where does one use neutral comparison?

Declarative specification

Arbitrarily mixing:

- Refl./Sym./Trans.
- Congruences
- Computation (β)
- Extensionality (η)

Type-directed algo.

Alternate

1. β-reduction to whnf
2. **Type**-directed η
3. Head congruences

"Untyped" algo.

Alternate

1. β-reduction to whnf
2. **Term**-directed η
3. Head congruences

Type-directed conversion

$$\frac{}{x : \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Type-directed conversion

$$\frac{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}{x\colon \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Type-directed conversion

$$\frac{\overline{}\quad x : \mathbb{N} \to \mathbb{N}, y : \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}{x : \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Type-directed conversion

$$\dfrac{\dfrac{\overline{x : \mathbb{N} \to \mathbb{N}, y : \mathbb{N} \vdash x\ y \sim x\ y : \mathbb{N}}}{x : \mathbb{N} \to \mathbb{N}, y : \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}}{x : \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Type-directed conversion

$$\dfrac{\dfrac{\dfrac{\dfrac{(x\colon \mathbb{N} \to \mathbb{N}) \in x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x \sim x : \mathbb{N} \to \mathbb{N}} \qquad \dfrac{\ldots}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash y \equiv y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \sim x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Type-directed conversion

$$\dfrac{\dfrac{\dfrac{\dfrac{(x\colon \mathbb{N} \to \mathbb{N}) \in x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x \sim x : \mathbb{N} \to \mathbb{N}} \qquad \dfrac{\dots}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash y \equiv y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \sim x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Term-directed conversion

$$\dfrac{\dfrac{}{x \sim x}}{x \equiv x}$$

Type-directed conversion

$$\dfrac{\dfrac{\dfrac{\dfrac{(x\colon \mathbb{N} \to \mathbb{N}) \in x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x \sim x : \mathbb{N} \to \mathbb{N}} \qquad \dfrac{\dots}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash y \equiv y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \sim x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N}, y\colon \mathbb{N} \vdash x\ y \equiv x\ y : \mathbb{N}}}{x\colon \mathbb{N} \to \mathbb{N} \vdash x \equiv x : \mathbb{N} \to \mathbb{N}}$$

Term-directed conversion

$$\dfrac{\dfrac{}{x \sim x}}{x \equiv x}$$

Why does this work?

Injectivity of neutral eliminators?

If $\Gamma \vdash n \equiv n' : T$ and n and n' are neutrals, then

- $n = x = n'$
- or $n = m\ u, n' = m'\ u'$ with $\Gamma \vdash m \equiv m' : \Pi\, x{:}\, A.B$ and $\Gamma \vdash u \equiv u' : A$
- or...

Injectivity of neutral eliminators?

If $\Gamma \vdash n \equiv n' : T$ and n and n' are neutrals, then

- $n = x = n'$
- or $n = m\, u$, $n' = m'\, u'$ with $\Gamma \vdash m \equiv m' : \Pi\, x{:}\, A.B$ and $\Gamma \vdash u \equiv u' : A$
- or…

Does not always hold!

$$x, y{:}\, (\mathbb{N} \to \mathbb{1}) \times \mathbb{1} \vdash x \equiv y : (\mathbb{N} \to \mathbb{1}) \times \mathbb{1}$$

Injectivity of neutral eliminators?

If $\Gamma \vdash n \equiv n' : T$ and n and n' are neutrals, then

- $n = x = n'$
- or $n = m\, u$, $n' = m'\, u'$ with $\Gamma \vdash m \equiv m' : \Pi\, x{:}\, A.B$ and $\Gamma \vdash u \equiv u' : A$
- or…

Does not always hold!

$$x, y{:}\, (\mathbb{N} \to \mathbb{1}) \times \mathbb{1} \vdash x \equiv y : (\mathbb{N} \to \mathbb{1}) \times \mathbb{1}$$

Neutral comparison is complete only at certain types

Injectivity of neutral eliminators?

If $\Gamma \vdash n \equiv n' : T$ and n and n' are neutrals, then

- $n = x = n'$
- or $n = m\ u$, $n' = m'\ u'$ with $\Gamma \vdash m \equiv m' : \Pi x{:}\, A.B$ and $\Gamma \vdash u \equiv u' : A$
- or…

Does not always hold!

$$x, y{:}\, (\mathbb{N} \to \mathbb{1}) \times \mathbb{1} \vdash x \equiv y : (\mathbb{N} \to \mathbb{1}) \times \mathbb{1}$$

Neutral comparison is complete only at certain types

Agda
Type-directed
Short path for neutral functions

Lean
Term-directed
Detect unit-like types

Rocq
Term-directed
Forbid unit-like types

Wrapping up

Verification is very useful!

- Bug finding
- Looking hard at the dark corners
- Uncover & understand assumptions hidden in implementation subtleties

Looking back

Verification is very useful!

- Bug finding
- Looking hard at the dark corners
- Uncover & understand assumptions hidden in implementation subtleties
- Documentation
- Reduce folklore
- And more

Verification is very useful!

- Bug finding
- Looking hard at the dark corners
- Uncover & understand assumptions hidden in implementation subtleties
- Documentation
- Reduce folklore
- And more

But: the cost is currently quite high…

Lots of cool things happening in the space:

AgdaCore	Agda-style pattern-matching + termination-checker
Lean4Lean	essentially the real kernel's code, with a dedicated program logic
MetaRocq	algebraic universes, nested inductive types...
λ□	whole ecosystem of verified compilations
And more	LogRel-Rocq, LogRel-MLTT/`graded-type-theory`, McTT, Liu & Weirich...

Looking forward

Lots of cool things happening in the space:

AgdaCore	Agda-style pattern-matching + termination-checker
Lean4Lean	essentially the real kernel's code, with a dedicated program logic
MetaRocq	algebraic universes, nested inductive types...
λ□	whole ecosystem of verified compilations
And more	LogRel-Rocq, LogRel-MLTT/`graded-type-theory`, McTT, Liu & Weirich...

Lowering the cost?

- Reusable libraries and insight
- Automation (AutoSubst/Sulfur)
- Forkable flagship projects

Looking forward

Lots of cool things happening in the space:

AgdaCore	Agda-style pattern-matching + termination-checker
Lean4Lean	essentially the real kernel's code, with a dedicated program logic
MetaRocq	algebraic universes, nested inductive types…
λ□	whole ecosystem of verified compilations
And more	LogRel-Rocq, LogRel-MLTT/`graded-type-theory`, McTT, Liu & Weirich…

Lowering the cost?

- Reusable libraries and insight
- Automation (AutoSubst/Sulfur)
- Forkable flagship projects

How do we bridge the gap with meta-theory-oriented formalisations?

We can (should!) separate **meta-theory** and **verification**

Find the right interfaces, from **synthetic methods** to **MetaRocq and beyond**

A lot is happening, **stay tuned**, or better: **come join us!**

We can (should!) separate **meta-theory** and **verification**

Find the right interfaces, from **synthetic methods** to **MetaRocq and beyond**

A lot is happening, **stay tuned**, or better: **come join us!**

Thank you!

Bibliography

[1] Matthieu Sozeau et al. 'Correct and Complete Type Checking and Certified Erasure for Coq, in Coq'. In: *Journal of the ACM* (Jan. 2025). DOI: 10.1145/3706056.

[2] Matthieu Sozeau, Meven Lennon-Bertrand and Yannick Forster. 'The Curious Case of Case: Correct & Efficient Representation of Case Analysis in Coq and MetaCoq'. Talk. 2022.

[3] Arthur Adjedj et al. 'Martin-Löf à la Coq'. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2024. DOI: 10.1145/3636501.3636951.

[4] Meven Lennon-Bertrand. 'What Does It Take to Certify a Conversion Checker?' In: *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*. 2025. DOI: 10.4230/LIPIcs.FSCD.2025.27.

[5] Andreas Abel, Joakim Öhman and Andrea Vezzosi. 'Decidability of Conversion for Type Theory in Type Theory'. In: *Proc. ACM Program. Lang.* POPL (Dec. 2017). DOI: 10.1145/3158111.

[6] Junyoung Jang et al. 'McTT: A Verified Kernel for a Proof Assistant'. In: *Proceedings of the ACM on Programming Languages* ICFP (Aug. 2025). DOI: 10.1145/3747511.

[7] Yiyun Liu, Jonathan Chan and Stephanie Weirich. 'Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories'. 2025. URL: https://github.com/yiyunliu/mltt-consistency.