WITS 2022 Workshop

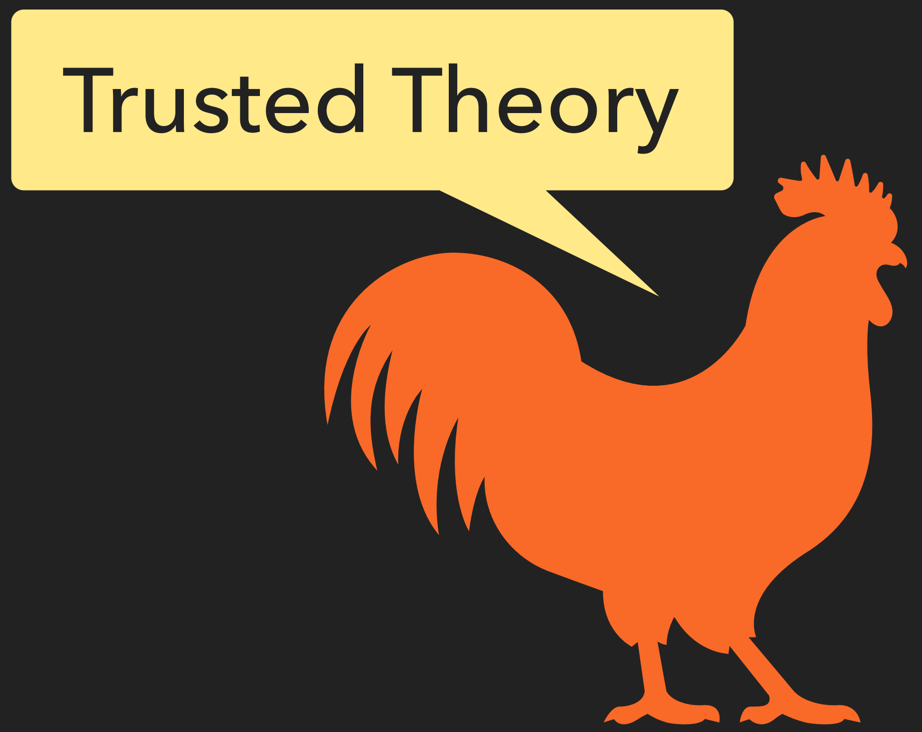
The Curious Case of Case

Correct & efficient representation of case analysis in Coq and MetaCoq

Matthieu **Sozeau** Meven **Lennon-Bertrand** Yannick **Forster**

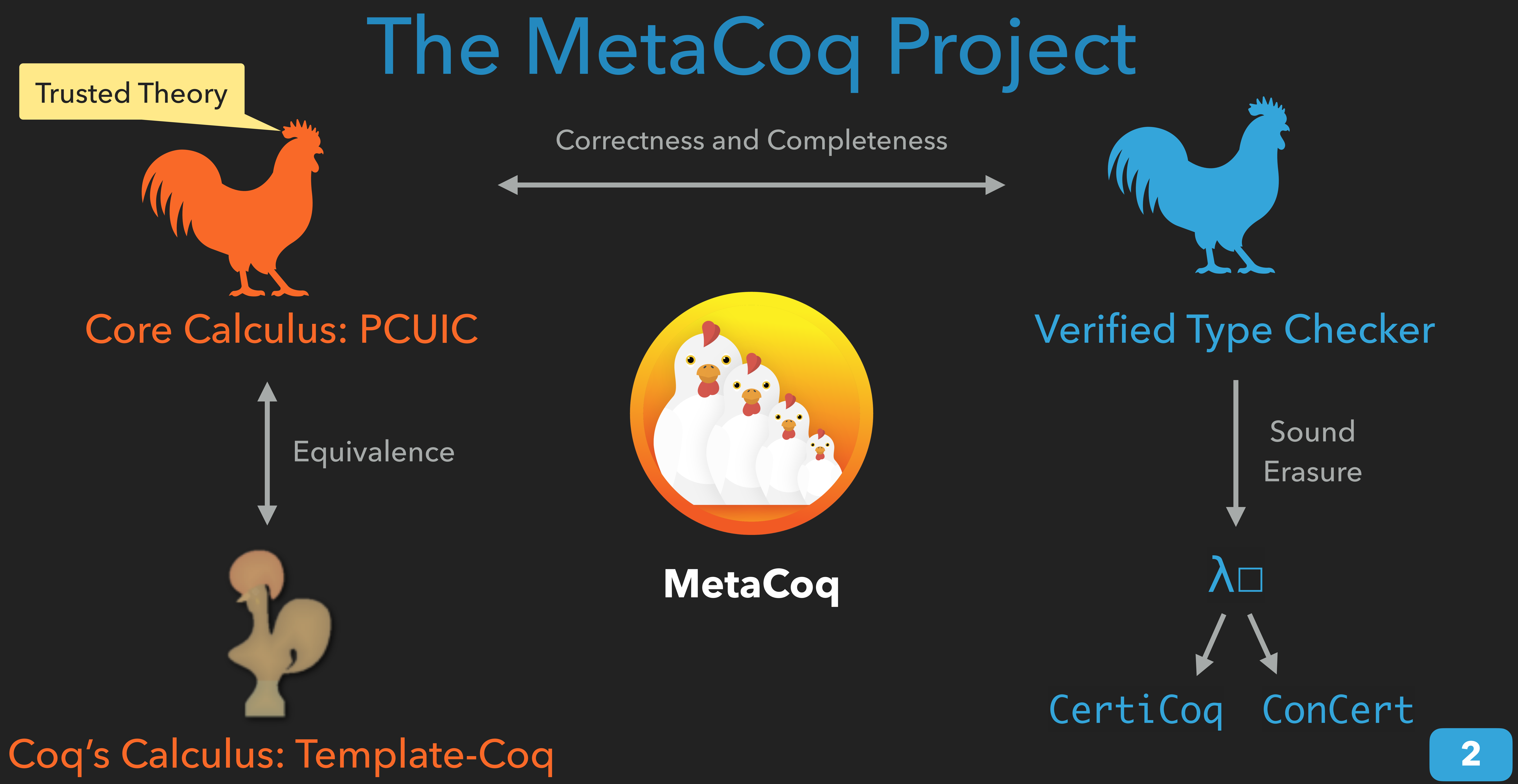
Inria & Université de Nantes

Our Goal: Improving Trust



Ideal Coq

Implemented Coq


The MetaCoq Project
Trusted Theory
Correctness and Completeness
Core Calculus: PCUIC
Verified Type Checker
Equivalence
MetaCoq
Sound
Erasure
λ□
Coq's Calculus: Template-Coq
CertiCoq
ConCert

A little success story

Spec/Proof/Program co-design for the new `match` representation

(CEP #34 by H. Herbelin, Coq PR #13563 by P.M. Pédrot)

```
Σ ; Γ ⊢ p : eq@{i'} A' t y
Σ ; Γ ⊢ fun y e => P : (forall y, eq@{i} A t y -> Type)
Σ ; Γ ⊢ b : P t eq_refl
i' = i     A <= A'
────────────────────────────────────────────────────────
Σ ; Γ ⊢ match p as e in eq _ _ y return P y e with
        | eq_refl => b end : P y p
```

Confusion in the kernel:

```
(fun (y : A) (e : eq@{i} A x y) => P y e)
                    ≢
(fun (y : A') (e : eq@{i'} A' x y) => P y e)
```

A little success story

- MetaCoq bidirectional typing completeness proof failure => typechecking of case on cumulative inductive types is incomplete
- Subject reduction failure in Coq (Coq issue #13495)
- Quick & dirty fix requires **strengthening**, not provable by induction

```
Σ ; Γ ⊢ _ : forall (y : A), eq@{i} A t y -> Type
```

does not imply `Σ ; Γ ⊢ t : A` without strengthening
(easy to forget when you're just implementing!)

A little success story

$$\frac{\begin{array}{l}\Sigma \,;\, \Gamma \vdash A : \mathrm{Type} \qquad \Sigma \,;\, \Gamma \vdash t : A \\ \Sigma \,;\, \Gamma, x : A, e : \mathtt{eq@\{i\}}\ A\ t\ x \vdash P : \mathrm{Type} \\ \Sigma \,;\, \Gamma \vdash p : \mathtt{eq@\{i\}}\ A\ t\ u \\ \Sigma \,;\, \Gamma \vdash b : P[t/x]\ \mathtt{eq\_refl}\end{array}}{\begin{array}{l}\Sigma \,;\, \Gamma \vdash \mathtt{match}\ p\ \mathtt{as}\ e\ \mathtt{in}\ \mathtt{eq@\{i\}}\ A\ t\ x\ \mathtt{return}\ P\ \mathtt{with} \\ \qquad \mathtt{|\ eq\_refl\ =>}\ b\ \mathtt{end} : P\ u\ p\end{array}}$$

- Case carries parameters (A, t) and a universe instance (i) separately + bindings names for the context. Derivation of the predicate and branch contexts on the fly.
- Completeness holds, reflects the high-level user syntax more closely
- Keeps cumulativity orthogonal to this rule
- Clear information flow in the bidirectional version

Bidirectional Type-Checking for the Win!

- **Bidirectional derivations** are syntax directed:
  Compressed and localised conversion rules.
- **Trivialises** correctness and completeness of type inference
- **Principality** follows from correctness and completeness of bidirectional typing w.r.t. “undirected” typing
- **Completeness** proof requires **injectivity** of type constructors
- **Correctness** proof requires **transitivity** of conversion
- **Strengthening** follows directly

WIP Correct and Complete type checking for PCUIC

The impact on MetaCoq

- Due to let-bindings in contexts, we must carry the "canonical context" of predicates and branches in PCUIC to still have well-behaved recursive definitions of renaming/substitution etc.

- This in turn requires to consider only well-scoped terms for the theory of reduction which was previously valid even on ill-scoped terms.

- Equivalence with Coq's theory where only binding names are carried => reason on the wellformedness of the global environment from which the canonical context is built.

- Non-trivial translation before erasure to expand let-bindings in branches.

Takeaways

- Always define a bidirectional version of your system:
  it will keep you honest
- "Minor" implementation changes can rely on subtle assumptions about the theory or require non-trivial reorganisations of the metatheory: they can no longer go unnoticed if you formalise it!
- The overhead of formalising is high but can catch bugs earlier.
  The initial investment pays off when we want to extend the language (c.f. Meven Lennon-Bertrand's talk on eta-conversion)