MetaRocq: Metaprogramming and Mechanization of Rocq in Rocq

EuroProofNET WG 3 Meeting
September 16th 2025

Matthieu Sozeau
Inria & LS2N, University of Nantes

joint work with

Abhishek **Anand** — Bedrock Systems, Inc
Danil **Annenkov** — University of Copenhagen
Andrew **Appel** — Princeton University
Simon **Boulier** — University of Nantes
Cyril **Cohen** — Inria
Yannick **Forster** — Inria
Joomy **Korkut** — Princeton University
Jason **Gross** — MIRI
Meven **Lennon-Bertrand** — University of Nantes
Gregory **Malecha** — Bedrock Systems, Inc
Jakob **Botsch Nielsen** — University of Copenhagen
Zoe **Paraskevopoulou** — University of Athens
Nicolas **Tabareau** — Inria & LS2N
Théo **Winterhalter** — Inria & LS2N

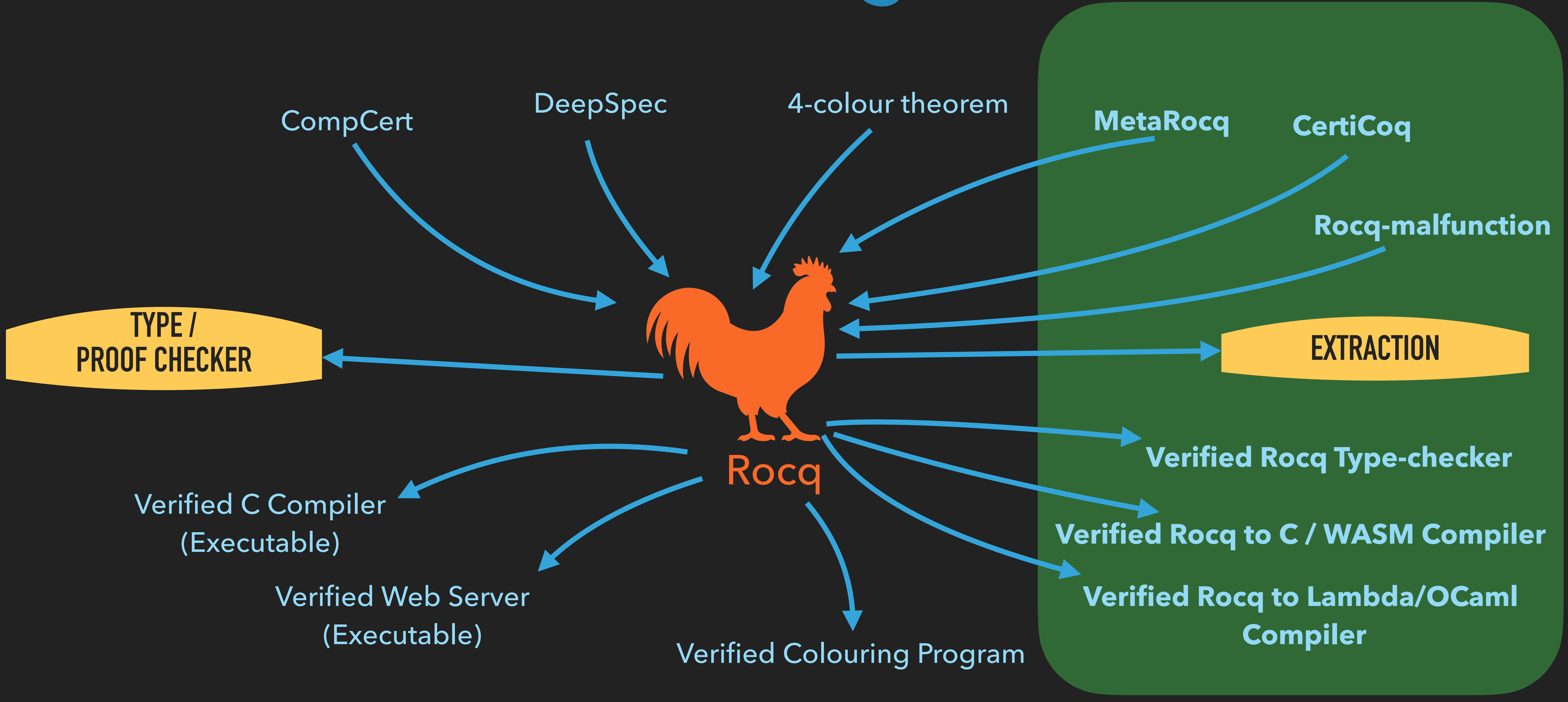
The MetaRocq Team

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

Disclaimers

An experience report on a meta-mathematical /meta-theoretical library

Setting

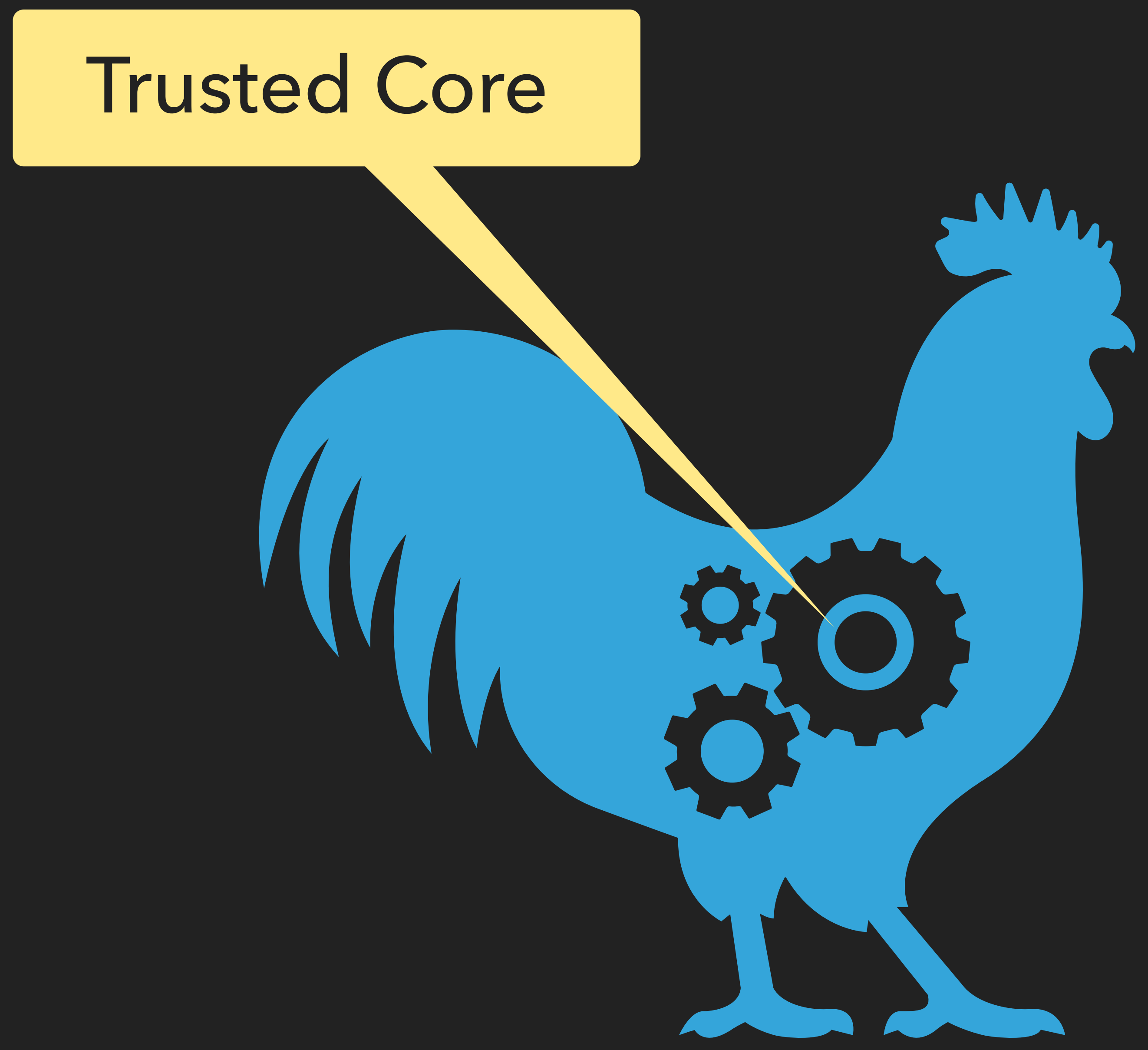

CompCert
DeepSpec
4-colour theorem
MetaRocq
CertiCoq
Rocq-malfunction
TYPE / PROOF CHECKER
EXTRACTION
Rocq
Verified C Compiler (Executable)
Verified Web Server (Executable)
Verified Colouring Program
Verified Rocq Type-checker
Verified Rocq to C / WASM Compiler
Verified Rocq to Lambda/OCaml Compiler

What do you trust?

A Dependent Type Checker for PCUIC
(18kLoC, 35+ years)

- (Co-)Inductive Families w/ Guard Checking
- Universe Cumulativity and Polymorphism
- ML-style Module System
- KAM, VM and Native Conversion Checkers
- Extraction if you extract your programs

+ OCaml's Compiler and Runtime



Implemented Rocq

The Reality

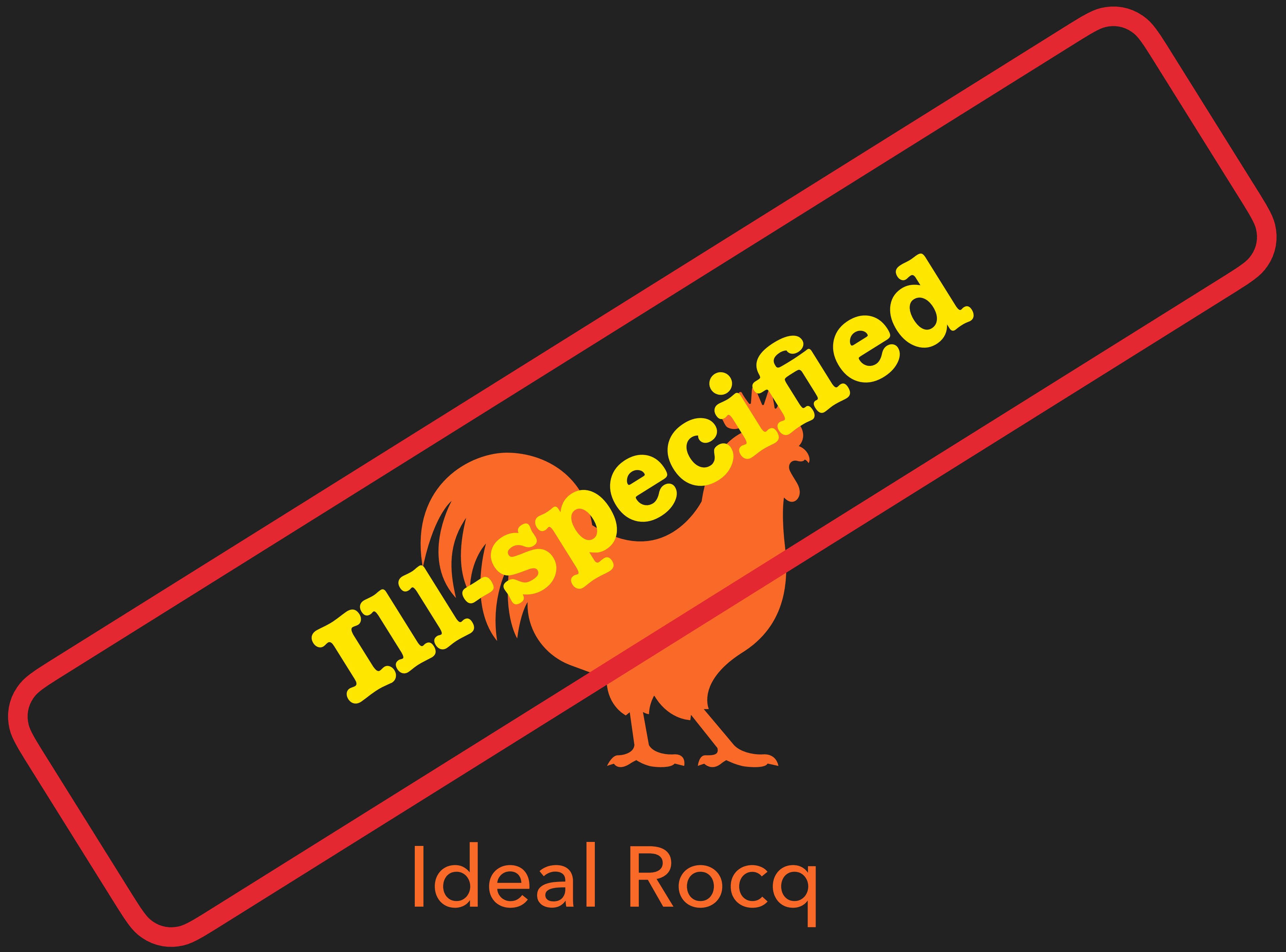
Ill-specified

Ideal Rocq

Buggy

Implemented Rocq

Reality Check



Ideal Rocq

- Reference Manual is semi-formal and partial
- “One feature = n papers/PhDs” where `n : fin 5` e.g. modules, universes, eta-conversion, guard condition, SProp….
- “Discrepancies” with the OCaml implementation
- Combination of features not worked-out in detail. E.g. cumulative inductive types + let-bindings in parameters of inductives???

Reality Check

```
354 lines (314 sloc)  |  16.7 KB

Preliminary compilation of critical bugs in stable rel
==========================================================
WORK IN PROGRESS WITH SEVERAL OPEN QUESTIONS


To add: #7723 (... universe polymorphism), #769

Typing constructions

  component: "match"
  summary: substitution missing in the body of a let
  introduced: ?
  impacted released versions: V8.3-V8.3pl2, V8.4-V8.4pl
  impacted development branches: none
  impacted coqchk versions: ?
  fixed in: master/trunk/v8.5 (e583a79b5, 22 Nov 2015,
  found by: Herbelin
```

In the news last week…

```
component: modules, primitive types
summary: Primitives are incorrectly considered convertible to anything by module subtyping
introduced: 8.11
impacted released versions: V8.11.0-V8.18.0
impacted coqchk versions: same
fixed in: V8.19.0
found by: Gaëtan Gilbert
GH issue number: #18503
exploit: see issue
risk: high if there is a Primitive in a Module Type, otherwise low
```

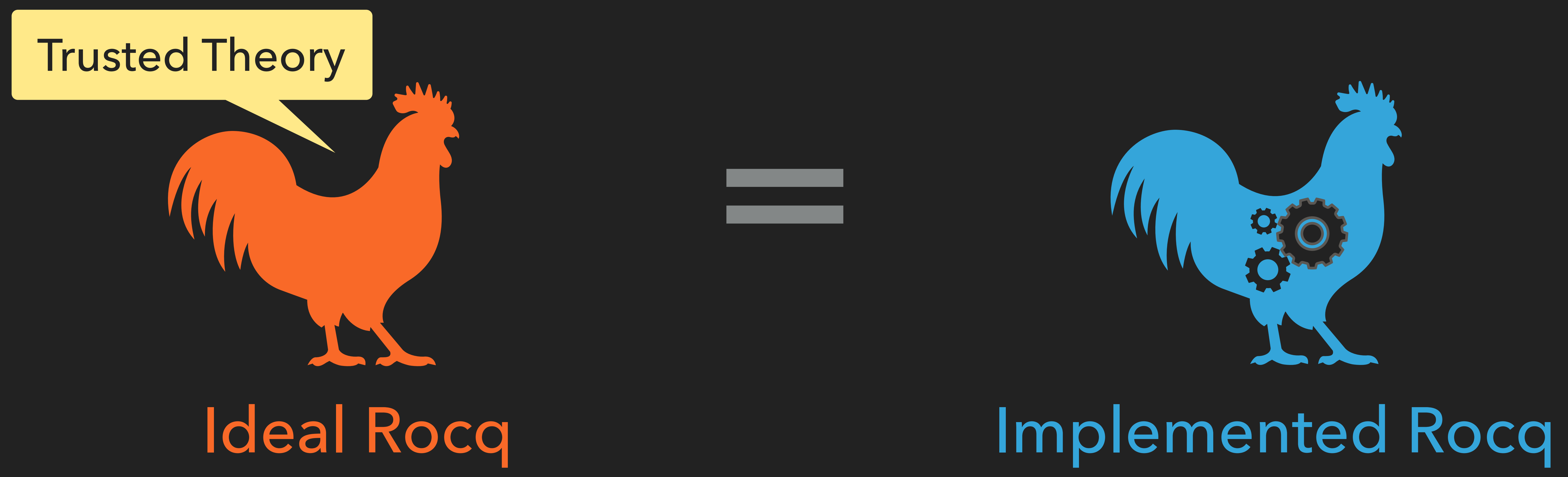
```
-        | Primitive _ | Undef _ | OpaqueDef _ -> cst
-        | Def c2 ->
-          (match cb1.const_body with
-            | Primitive _ | Undef _ | OpaqueDef _ -> error NotConvertibleBodyField
-            | Def c1 ->
-              (* NB: cb1 might have been strengthened and appear as transparent.
-                 Anyway [check_conv] will handle that afterwards. *)
-              check_conv NotConvertibleBodyField cst poly CONV env c1 c2))
+        | Undef _ | OpaqueDef _ -> cst
+        | Primitive _ -> error NotConvertibleBodyField
+        | Def c2 ->
+          (match cb1.const_body with
+            | Primitive _ | Undef _ | OpaqueDef _ -> error NotConvertibleBodyField
+            | Def c1 ->
+              (* NB: cb1 might have been strengthened and appear as transparent.
+                 Anyway [check_conv] will handle that afterwards. *)
+              check_conv NotConvertibleBodyField cst poly CONV env c1 c2))
```

The situation today

Ideal Rocq

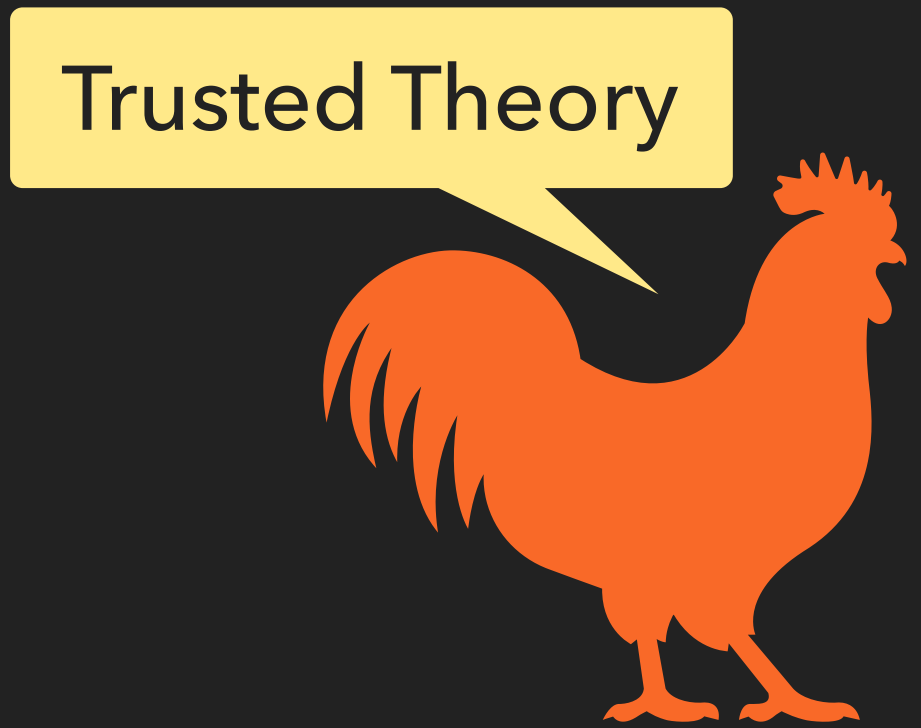
Implemented Rocq

Our Goal: Improving Trust

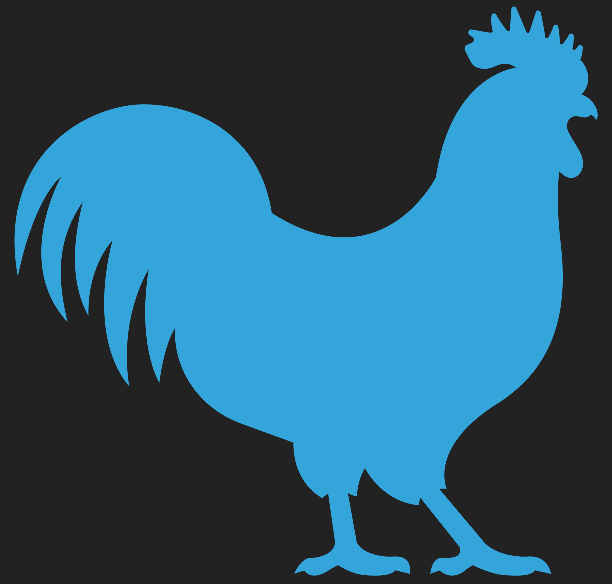


Rocq in MetaRocq

Verified metatheory,
correct implementations



Rocq's Calculus PCUIC

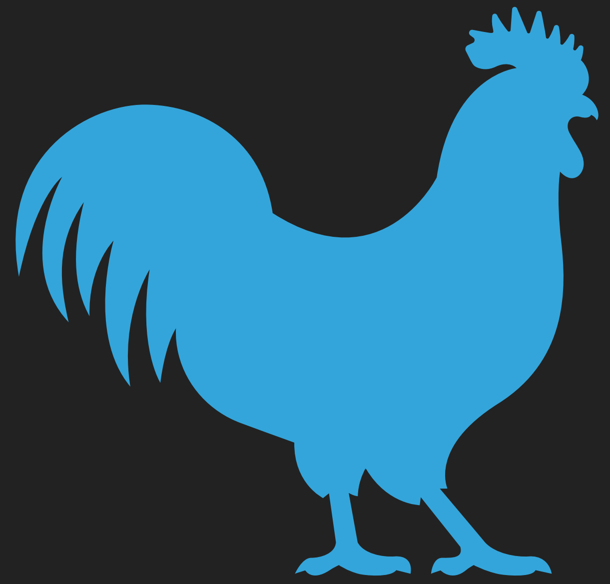


Verified Rocq

in

MetaRocq
Formalization of
Rocq in Rocq

in



Implemented Rocq

Together with Verified Extraction


```
MetaRocq Check infer.
```

POPL'20, JACM'25

Verified ℰ

```
MetaRocq Compile infer.
```

PLDI'24

Outline

I. A tour of MetaRocq: metaprogramming, meta-theory and verified implementation of Rocq in Rocq

II. Formalization challenges

Contents of MetaRocq

- Template-Rocq: metaprogramming in Rocq (20kLoC)
- PCUIC: meta-theory of Rocq in Rocq (150kLoC)
- A Verified Rocq type-checker (20kLoC)
- A Verified Rocq type-and-proof erasure procedure (45kLoC)
- Quotation: formalization of Löb's theorem (6kLoC)
- A verified extraction to OCaml (20kLoC)
- Total (w/ utils) 250kLoC –- Extracted OCaml ~ 100kLoC

A bit of history

- Template-Coq (Malecha, 2014): a bare-bones library for reflection of Coq terms into Coq itself: i.e. the AST of Coq (~ Expr in Lean) and minimal meta-programming support.
- Used in the CertiCoq project (2015): verified compiler from Coq to C, using a trusted, unverified erasure procedure to λ-calculus, extended meta-programming support
- 2016-2022: MetaRocq: meta-theory, checkers and erasure
- 2020-2024: Verified erasure and extraction to OCaml

MetaRocq in Practice
A meta-programming library

DEMO!

Rocq’s Type Theory: PCUIC

The (Predicative) Polymorphic Cumulative Calculus of (Co-)Inductive Constructions

What we represent...

```
fix vrev {A : Type@{i}} {n m : nat} (v : vec@{i} A n) (acc : vec@{i} A m) :=
  match v in vec _ n return vec@{i} A (n + m) with
  | vnil          ⇒ acc
  | vcons a n v’ ⇒
      let idx := S n + m in
      coerce (vec A) idx (e : n + S m = idx) (vrev v’ (vcons a m acc))
  end.
```

```
vrev_term : term :=
tFix [{|
  dname := nNamed "vrev" ;
  dtype := tProd (nNamed « A") (tSort (Universe.make'' (Level.Level "Top.160", false) []))
    (tProd (nNamed "n") (tInd {| inductive_mind := "Rocq.Init.Datatypes.nat";
      inductive_ind := 0 |} [])
    (tProd (nNamed "m") (tInd {| ...
```

What we represent…

```
fix vrev {A : Type@{i}} {n m : nat} (v : vec@{i} A n) (acc : vec@{i} A m) :=
  match v in vec _ n return vec@{i} A (n + m) with
   | vnil          ⇒ acc
   | vcons a n v’  ⇒
      let idx := S n + m in
      coerce (vec A) idx (e : n + S m = idx) (vrev v’ (vcons a m acc))
  end.
```

Specification

Example: Reduction

DEFINITIONS IN CONTEXTS

$$\frac{(x : T := t) \in \Gamma}{\Gamma \vdash x \to t}$$

GENERAL SUBSTITUTION

$$\frac{}{\Gamma \vdash \texttt{let } x : T := t \texttt{ in } b \to b'[x := t]}$$

STRONG REDUCTION

$$\frac{\Gamma, x : T := t \vdash b \to b'}{\Gamma \vdash \texttt{let } x : T := t \texttt{ in } b \to \texttt{let } x : T := t \texttt{ in } b'}$$

Meta-Theory

Structures

```
term, t, u ::=
  | Rel (n : nat) | Sort (u : universe) | App (f a : term) …

global_env, Σ ::= []
  | Σ , (kername × InductiveDecl idecl)
  | Σ , (kername × ConstantDecl cdecl)

global_env_ext ::= (global_env × universes_decl)

Γ ::= []
   | Γ , aname : term
   | Γ , aname := t : u
```

(global environment)

(global environment with universes)

(local environment)

Meta-Theory

Judgments

$\Sigma \,;\, \Gamma \vdash t \to u,\ t \to^* u$

One-step reduction and its reflexive transitive closure (and many other variants)

$\Sigma \,;\, \Gamma \vdash t =_\alpha u,\ t \leq_\alpha u$

α-equivalence + equality or cumulativity of universes

$\Sigma \,;\, \Gamma \vdash T = U,\ T \leq U$

Untyped conversion and cumulativity

$\iff T \to^* T' \wedge U \to^* U' \wedge T' \leq_\alpha U'$

$\Sigma \,;\, \Gamma \vdash t : T$

Typing

`wf Σ, wf_local Σ Γ`

Well-formed global and local environments

Basic Meta-Theory

Structural Properties

- Traditional de Bruijn lifting and substitution operations as in Rocq
- Show that σ-calculus operations simulate them (à la Autosubst) :

```
ren : (nat -> nat) -> term -> term
inst : (nat -> term) -> term -> term
```

- Still useful to keep both definitions
- Weakening and Substitution from renaming and instantiation theorems
- Easy to lift to strengthening/exchange lemmas

Universes

```
universe ::= Prop | SProp
  | Type (ne_sorted_list (universe_level * nat)).
```

Typing `Σ ; Γ ⊢ tSort u : tSort (Universe.super u)`

No distinction of *algebraic* universes: more uniform than current Rocq, similar to Agda

```
universe_constraint ::=
  universe_level × ℤ × universe_level.     (u + x ≤ v)
```

Specification Global set of consistent constraints, satisfy a valuation in $\mathbb{N}$.

Universes

Basic Meta-Theory

Global environment weakening

Monotonicity of typing under context extension: universe consistency is monotone.

Universe instantiation

Easy, de Bruijn level encoding of universe variables (no capture)

Checking and satisfiability implementations

Longest simple paths in the graph generated by the constraints ϕ, with source `lSet`

```
∀ l, lsp ϕ l l = 0 ⟺ satisfiable ϕ (λ l, lsp lSet l)
```

Meta-Theory

The path to subject reduction

Validity

$$\frac{\Sigma \,;\, \Gamma \vdash t : T}{\Sigma \,;\, \Gamma \vdash T : \mathtt{tSort}\ s}$$

Requires **transitivity** of conversion/cumulativity

Context Conversion

$$\frac{\Sigma \,;\, \Gamma \vdash t : T \qquad \Sigma \vdash \Delta \leq \Gamma}{\Sigma \,;\, \Delta \vdash t : T}$$

More generally, context **cumulativity (contravariant)**

Subject Reduction

$$\frac{\Sigma \,;\, \Gamma \vdash t : T \qquad \Sigma \,;\, \Gamma \vdash t \rightarrow u}{\Sigma \,;\, \Gamma \vdash u : T}$$

Relies on **injectivity** of **type constructors**, a consequence of **confluence**

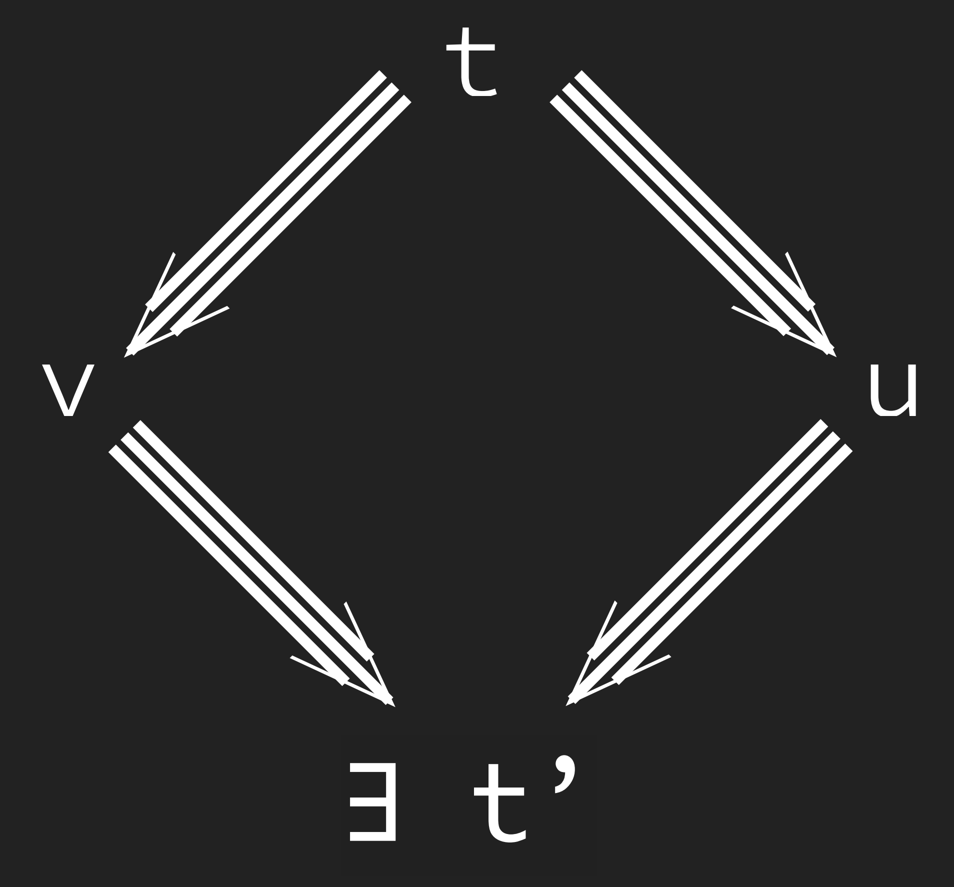
Confluence

The traditional way

$\Sigma \,,\, \Gamma \vdash t \Rrightarrow u$ One-step **parallel** reduction

À la Tait-Martin-Löf/Takahashi:

Diamond for $\Rrightarrow$



“Squash” lemma

$\_ \to \_$

$\subset \_ \Rrightarrow \_$

$\subset \_ \to^{*} \_$

Confluence

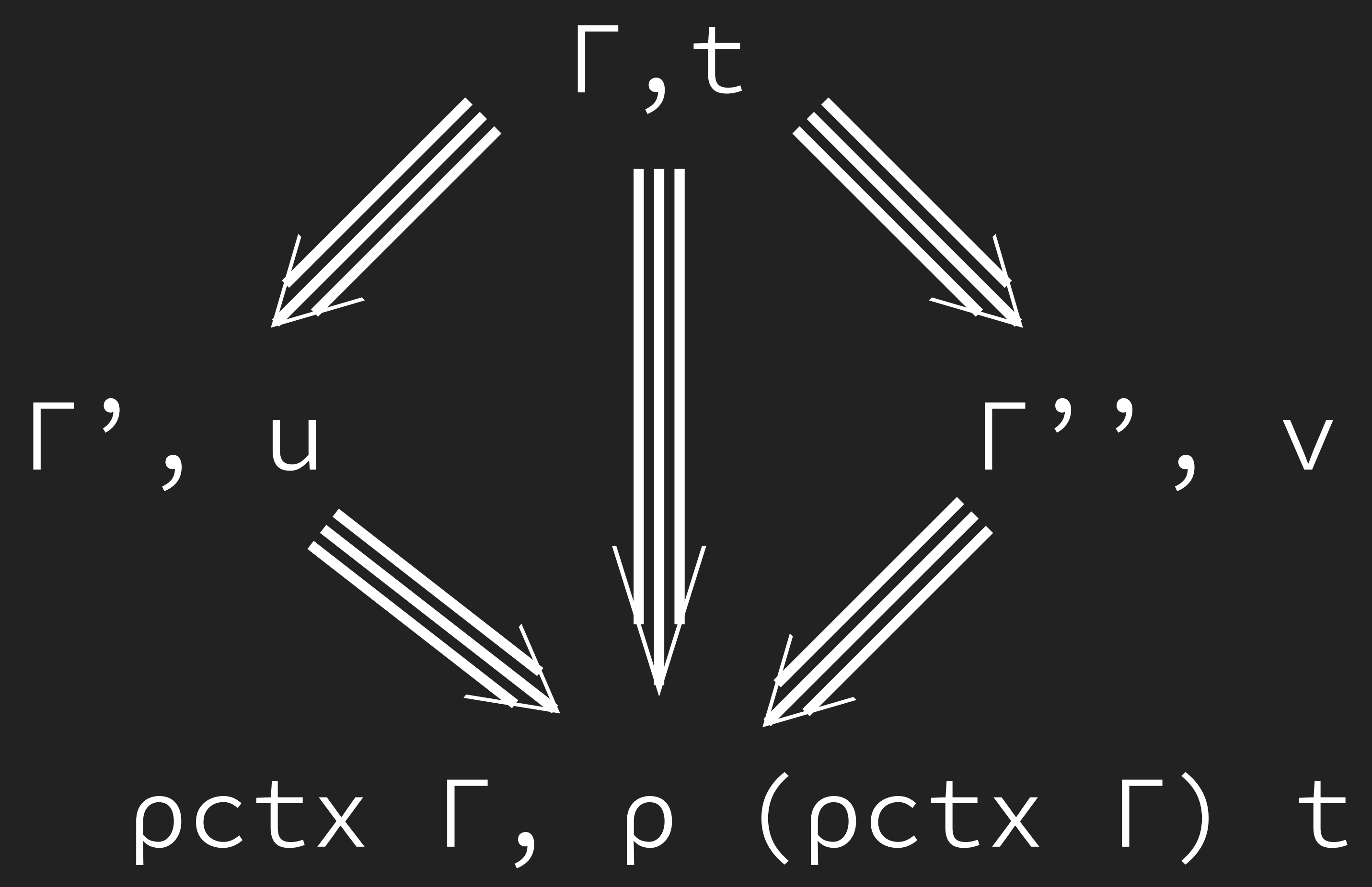
For a theory with definitions in contexts

```
Σ ⊢ Γ, t ⇛ Δ, u
```

One-step parallel reduction, including reduction in contexts.

```
Σ ⊢ Γ, x := t ⇛ Δ, x := t’   Σ ⊢ (Γ, x := t), b ⇛ (Δ, x := t’), b’
─────────────────────────────────────────────────────────────────────
     Σ ⊢ Γ, (let x := t in b) ⇛ Δ, (let x := t’ in b’)
```

```
ρ : context -> term -> term
ρctx : context -> context
```



Trusted Theory Base

Assumptions

- Typing, reduction and cumulativity: ~ 1kLoC (verified and testable)
- **Oracles for guard conditions**
  `check_fix : global_env → context → fixpoint → bool`
  + preservation by renaming/instantiation/equality/reduction
- WIP Rocq implementation of the guard/productivity checkers, and justification of it (Lamiaux, Forster, Sozeau, Tabareau)

Verifying a Type-Checker

Conversion

Objective

Input

$u : A$

$v : B$

Output

$(u \equiv v) + (u \not\equiv v)$

Conversion

Objective

Input

u : A

v : B

Output

(u ≡ v) + (u ≢ v)

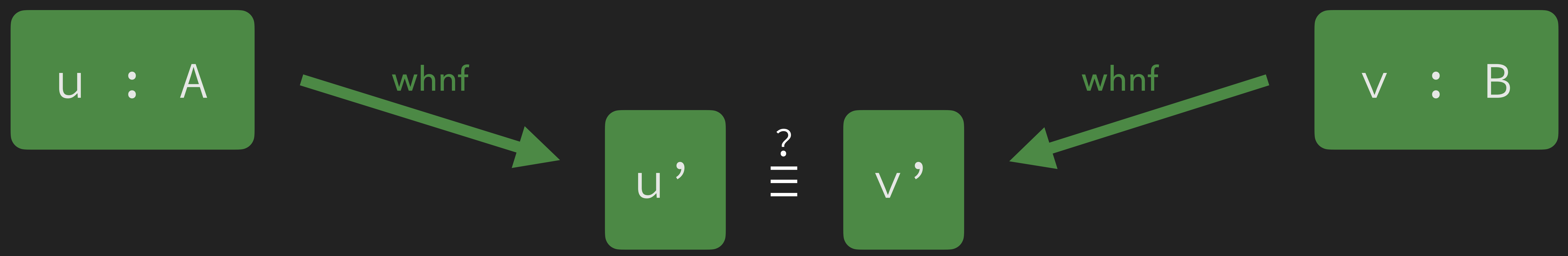
```
isconv :
  ∀ Σ Γ (u v A B : term),
    (Σ ; Γ ⊢ u : A) ->
    (Σ ; Γ ⊢ v : B) ->
    (Σ ; Γ ⊢ u ≡ v) +
    (Σ ; Γ ⊢ u ≡ v -> ⊥)
```

Conversion

Algorithm

Conversion

Algorithm



Conversion

Algorithm

u : A —whnf→ u'

v : B —whnf→ v'

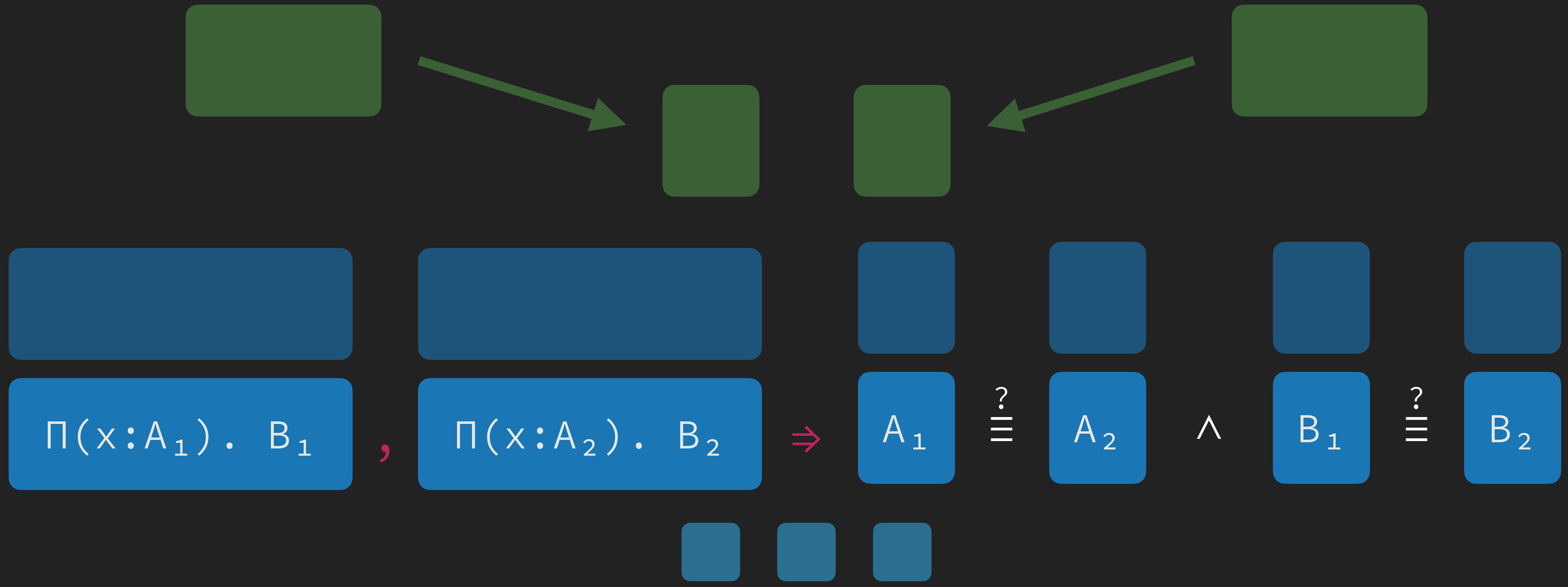
match $u' \stackrel{?}{\equiv}$, v' with

$\lambda(x{:}A_1).\ t_1$, $\lambda(x{:}A_2).\ t_2$ ⇒ $A_1 \stackrel{?}{\equiv} A_2 \wedge t_1 \stackrel{?}{\equiv} t_2$

$\Pi(x{:}A_1).\ B_1$, $\Pi(x{:}A_2).\ B_2$ ⇒ $A_1 \stackrel{?}{\equiv} A_2 \wedge B_1 \stackrel{?}{\equiv} B_2$

Conversion

Completeness



Conversion

Completeness

$\Pi(x{:}A_1).\ B_1 \overset{?}{\equiv} \Pi(x{:}A_2).\ B_2 \Rightarrow A_1 \not\equiv A_2$

Conversion

Completeness

$\Pi(x:A_1).\ B_1 \stackrel{?}{\equiv} \Pi(x:A_2).\ B_2 \Rightarrow A_1 \not\equiv A_2$

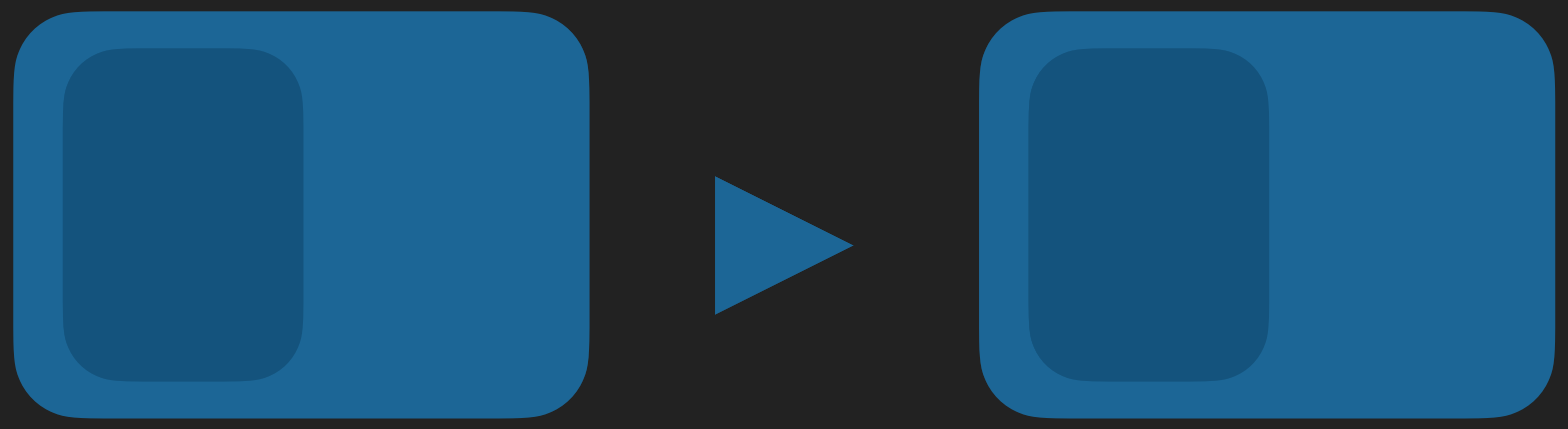
we conclude

$\Pi(x:A_1).\ B_1 \not\equiv \Pi(x:A_2).\ B_2$

using inversion lemmata and confluence

Weak head reduction

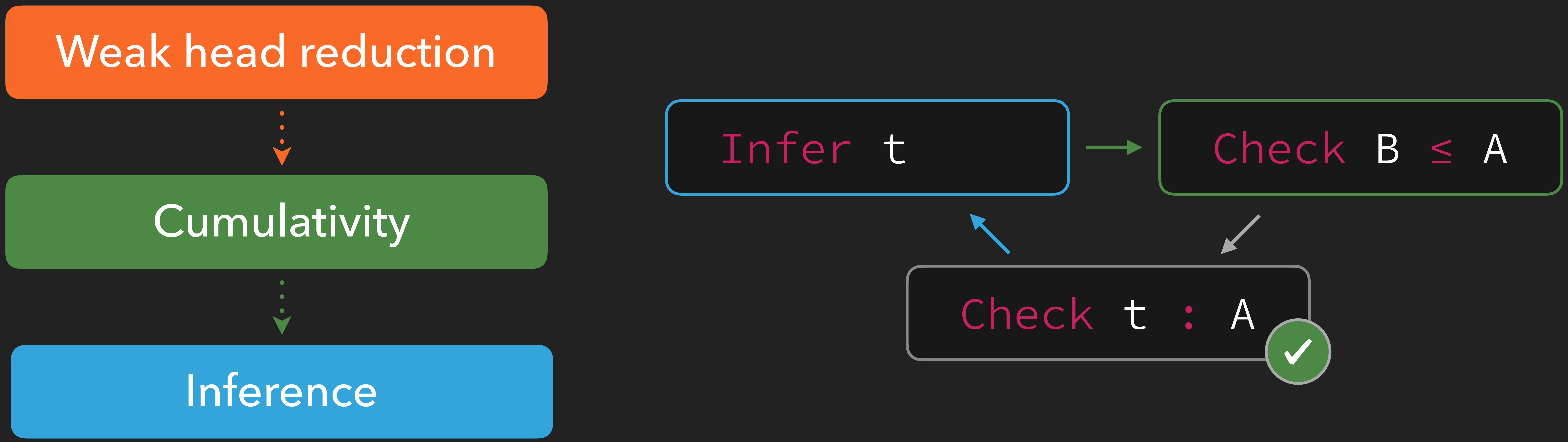
Termination



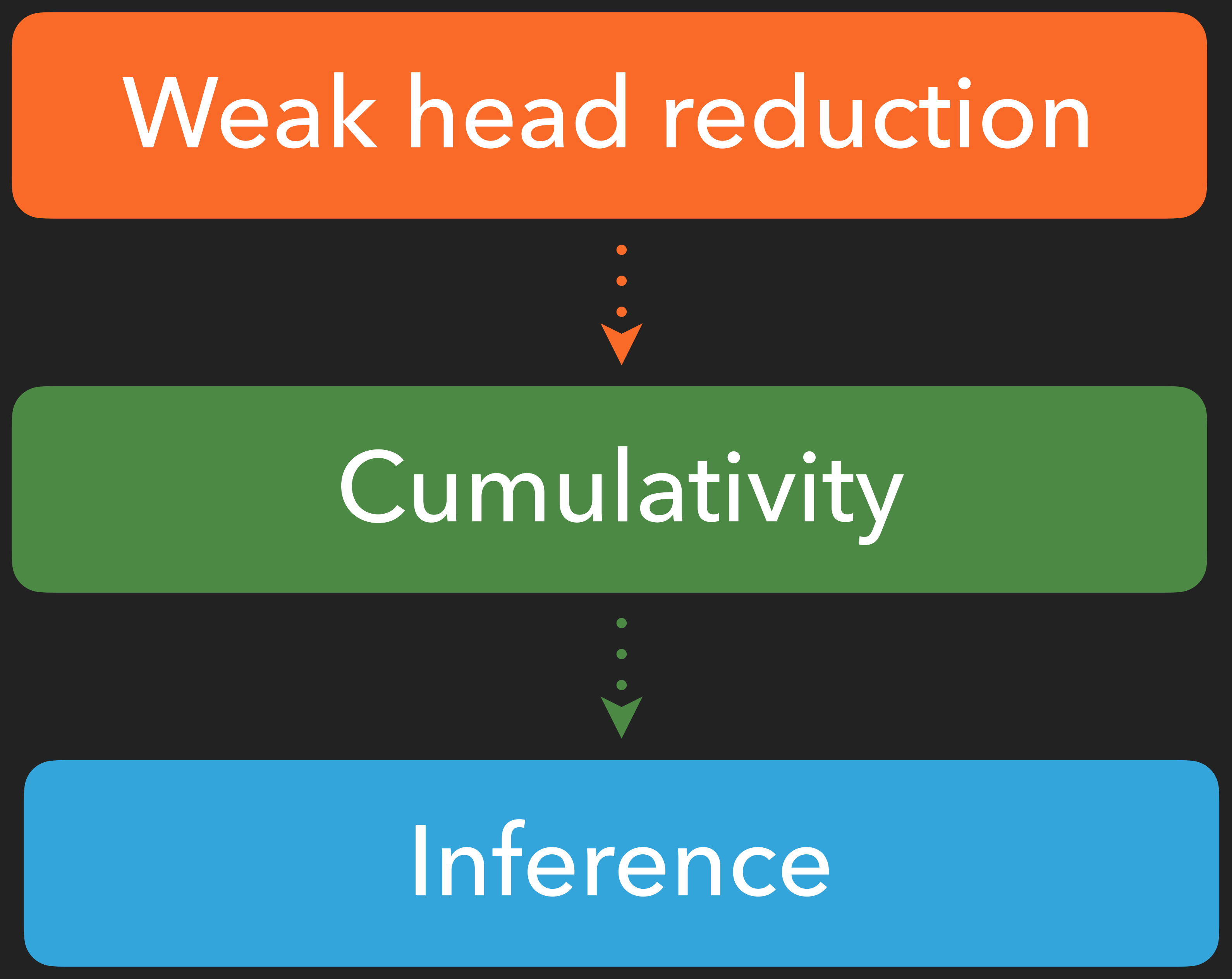
$$\langle\, u\ \pi_1\ ,\ \underbrace{\texttt{stack\_pos}\ u\ \pi_1}_{\texttt{pos}\ (u\ \pi_1)}\,\rangle > \langle\, v\ \pi_2\ ,\ \underbrace{\texttt{stack\_pos}\ v\ \pi_2}_{\texttt{pos}\ (v\ \pi_2)}\,\rangle$$

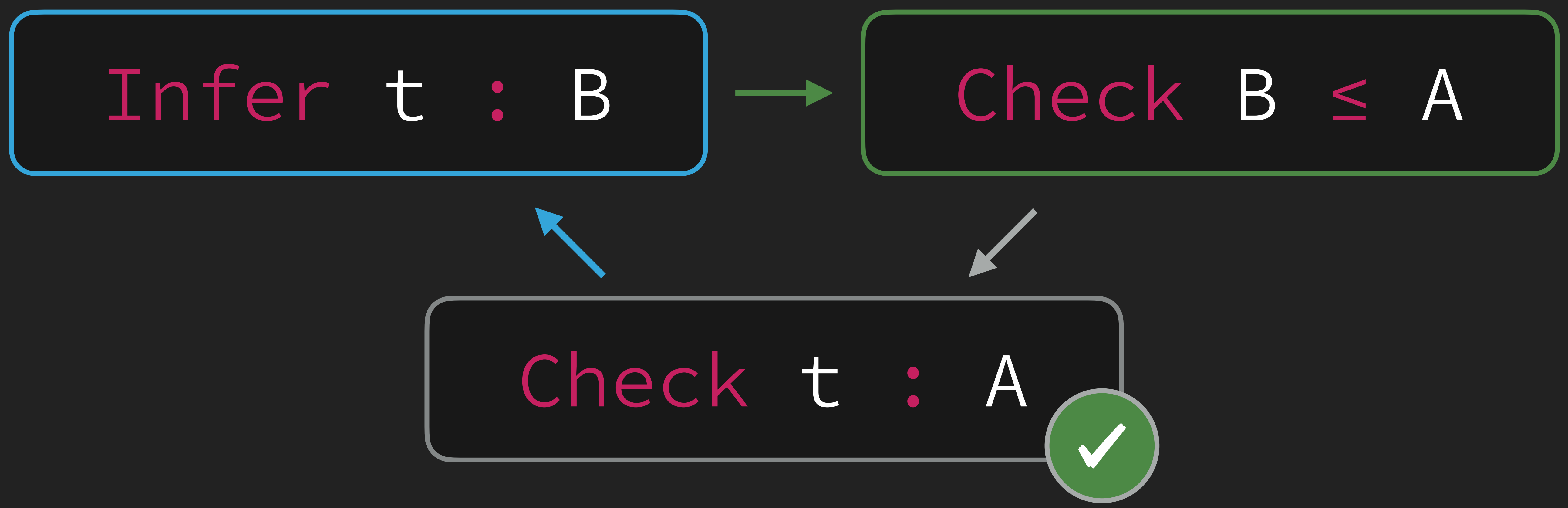
Dependent lexicographic order of `->` and an order on positions

Type Checking



Type Checking






```
MetaRocq Check foo.
```

Bidirectional Derivations

- General technique to show decidability of an inductively-defined relation/judgement
- Specify inputs and outputs of a relation:

```
    Σ ; Γ ⊢ t : T

    splits into mutually defined:

Inference
    Σ ; Γ ⊢ t > T
   (Σ, Γ, t well-formed inputs, T output)

and checking
    Σ ; Γ ⊢ t < T (Σ, Γ, t, T  well-formed inputs)
```

Bidirectional Type-Checking for the Win!

- Bidirectional derivations are syntax directed
- Trivialises correctness and completeness of type inference
- Principality follows from correctness and completeness of bidirectional typing w.r.t. “undirected” typing.
(some duplication of substitution / weakening etc… lemmas here)
- Completeness requires injectivity of type constructors
- Correctness requires transitivity of conversion
- Strengthening follows directly

Trusted Theory Base

Assumptions

```
Axiom normalisation :
  ∀ Σ Γ t, wf_global Σ -> wf_local Σ Γ ->
  welltyped Σ Γ t → Acc (cored Σ Γ) t.
```

- Strong Normalization
  Not provable thanks to Gödel's second incompleteness theorem.
- Consistency and canonicity follow easily.
- Used **exclusively** for termination of the conversion test

See Martin-Löf à la Coq (Adjedj et al, CPP'24) and "What Does It Take to Certify a Conversion Checker?" (Lennon-Bertrand, FSCD'25) for the state of the art!

Verifying Erasure

Erasure

At the core of the extraction mechanism:

$\mathcal{E}$: term → $\Lambda_{\square,\text{match},\text{fix},\text{cofix}}$

Erases non-computational content:

- Type erasure:

  $\mathcal{E}$ (t : Type) = □

- Proof erasure:

  $\mathcal{E}$ (p : P : Prop) = □

```
fix vrev {A : Type@{i}} {n m : nat} (v : vec A n)
(acc : vec A m) :=
  match v in vec _ n return vec A (n + m) with
  | vnil         ⇒ acc
  | vcons a n v' ⇒
      let idx := S n + m in
      coerce (vec A) idx (e : n + S m = idx)
        (vrev v' (vcons a m acc))
  end.
```

$\mathcal{E}$ (vrev) =

```
fix vrev n m v acc :=
  match v with
  | vnil         ⇒ acc
  | vcons a n v' ⇒
      let idx := S n + m in
      coerce □ idx □ (vrev v' (vcons a m acc))
  end.
```

Erasure

Singleton elimination principle

Erase propositional content used in computational content:

```
ℰ (match p in eq _ y with eq_refl ⇒ b end) = ℰ (b)
```

```
Definition coerce {A} {B : A -> Type) {x} (y : A)
(e : x = y) : P x -> P y :=
  match e with
  | eq_refl      ⇒ fun p ⇒ p
  end.

fix vrev n m v acc :=
  match v with
  | vnil          ⇒ acc
  | vcons a n v’ ⇒
      let idx := S n + m in
      coerce ☐ idx ☐ (vrev v’ (vcons a m acc))
  end.
```

Erasure

Singleton elimination principle

Erase propositional content used in computational content:

```
ℰ (match p in eq _ y with eq_refl ⇒ b end) = ℰ (b)
```

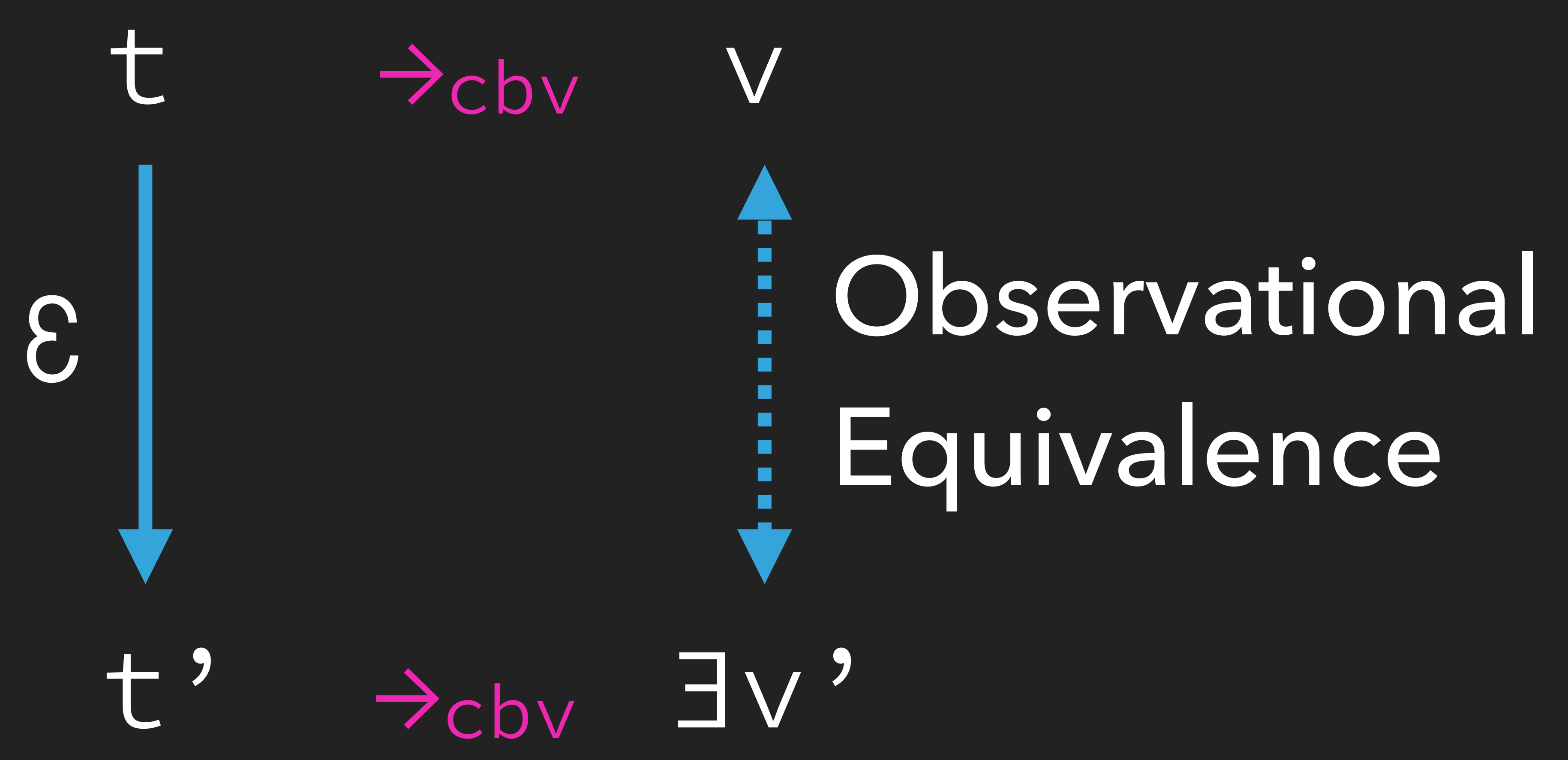
ℰ (coerce) ~

```
coerce x y := (fun p ⇒ p)
```

ℰ (vrev) ~

```
fix vrev n m v acc :=
  match v with
  | vnil          ⇒ acc
  | vcons a n v’ ⇒ vrev v’ (vcons a m acc)
  end.
```

Erasure Correctness




```
   ⊢ t : nat
=> ⊢ t → n /\ n irreducible (strong normalization)
=> ⊢ t → n : nat /\ n ∈ ℕ   (subject reduction and canonicity)
=> ⊢ t →cbv n /\ n ∈ ℕ       (standardisation)
=> ℰ (t) →cbv ℰ (n) = n      (erasure correctness +
                              extracted naturals are equivalent to naturals)
```

Erasure Correctness

First define a non-deterministic erasure relation, then define:

```
ℰ : ∀ Σ Γ t (wt : welltyped Σ Γ t) → EAst.term
```

Finally show that ℰ's graph is in the erasure relation. A few additional optimizations:

- Remove trivial cases on singleton inductive types in `Prop`
- Compute the dependencies of the erased term to erase only the computationally relevant subset of the global environment. I.e. remove unnecessary proofs the original term depended on.
- Inline projections, constructors as blocks (fully applied), unguarded fixpoint reduction

Verifying Extraction to OCaml

Malfunction & Rocq-malfunction

- AST of untyped OCaml terms (including refs, …)
  Using HOAS, tricky mutual fix point representation
- Compiler from malfunction to cmxs (ocaml object files), provided a trusted .mli interface is given.
- A reference interpreter ported to Rocq using a named variables variant of Λ□
- We derive a big-step operational semantics (with a heap and environment), producing malfunction values (closures, blocks for constructors, or primitive ints/floats), agreeing with the interpreter

Compiler Correctness

t $\rightarrow_{cbv}$ v (in $\Lambda^{\square,\text{match},\text{fix},\text{cofix}}$)

Rocq-malfunction

Observational Equivalence

t' $\rightarrow_{cbv}$ ∃v' (in OCaml/malfunction)

With Canonicity and SN:

$\vdash$ t : nat

=> $\vdash$ t $\rightarrow$ n : nat ($n \in \mathbb{N}$)

=> t $\rightarrow_{cbv}$ n : nat

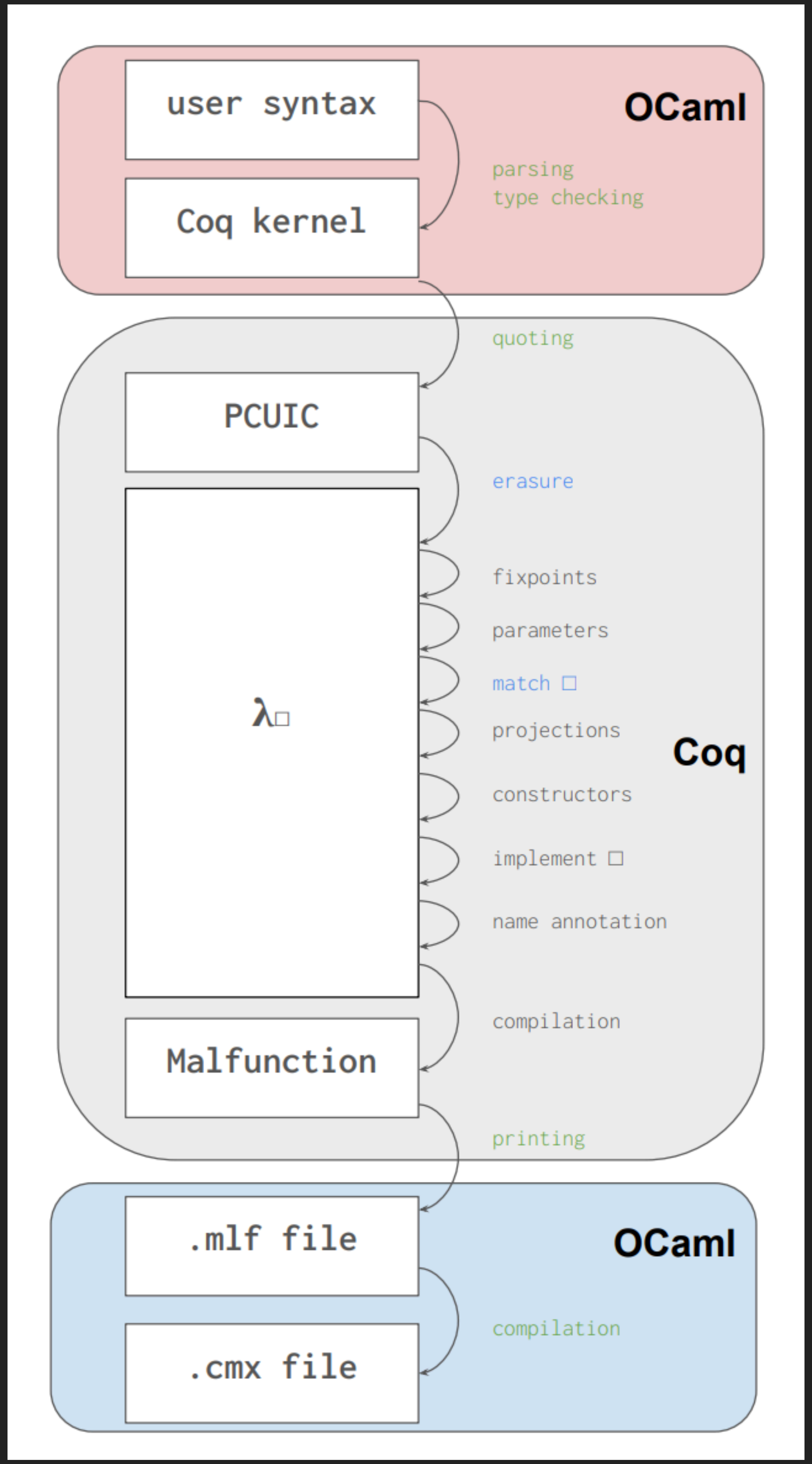
=> Rocq-malfunction (t) $\rightarrow_{cbv}$ n

Separate compilation

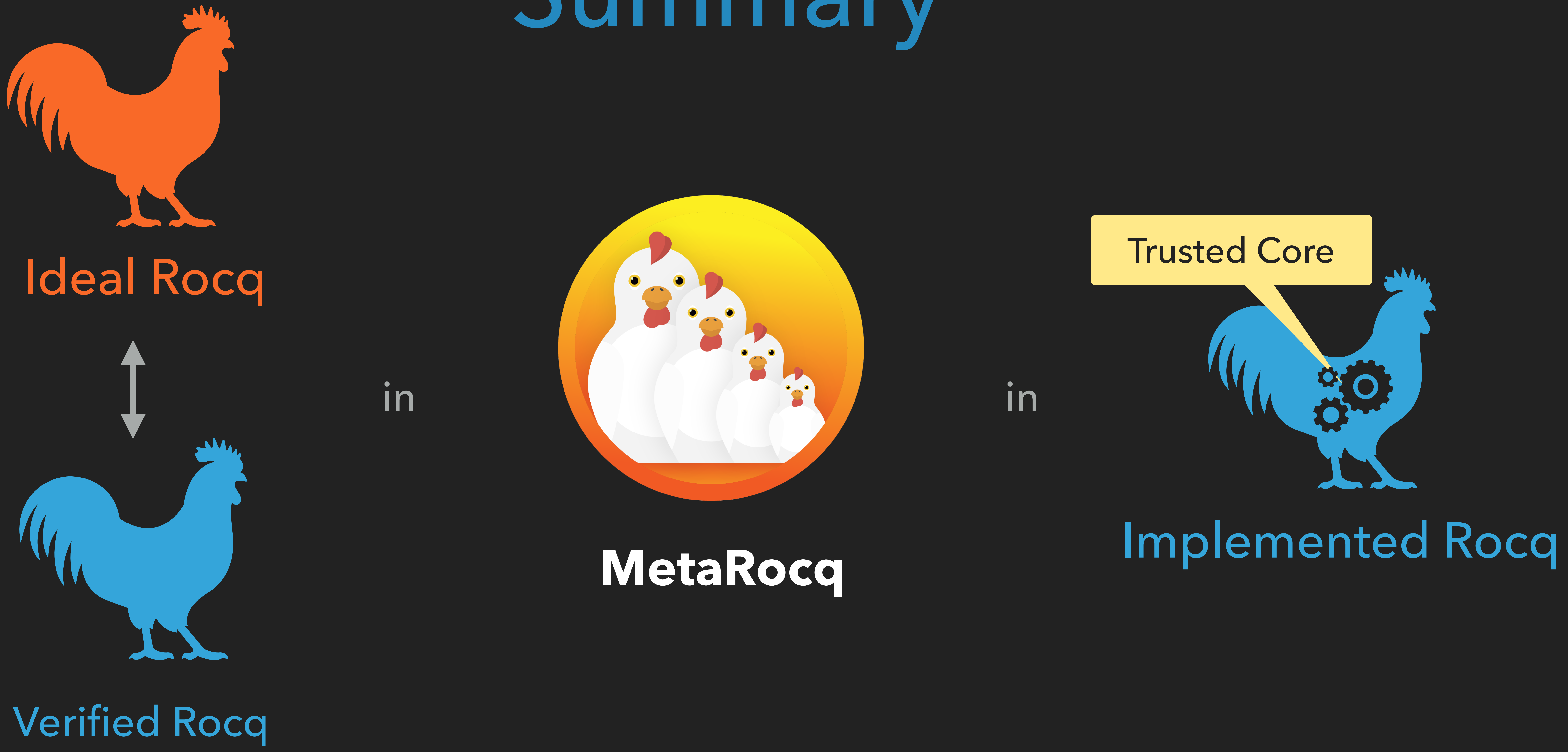
$$\frac{\vdash \texttt{t : nat -> nat} \qquad \vdash \texttt{u : nat} \qquad \texttt{t u} \rightarrow_{\texttt{cbv}} \texttt{n}}{\texttt{Mapply (Rocq-malfunction t) (Rocq-malfunction u)} \rightarrow_{\texttt{cbv}} \texttt{n}}$$

- Uses a step-indexed realisability semantics for the subset of ocaml types we consider (first-order datatypes)
- Requires to show that functions compiled from Rocq are pure (don't touch the heap).

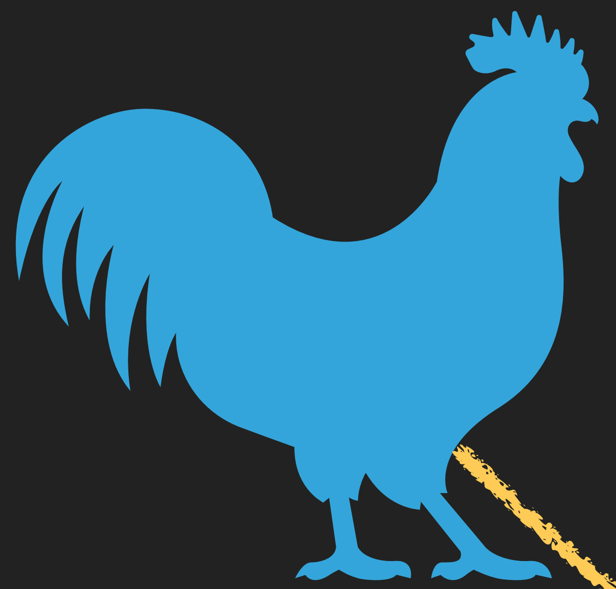
Verified Extraction Pipeline



Summary


Ideal Rocq
Verified Rocq
in
MetaRocq
in
Trusted Core
Implemented Rocq

Summary



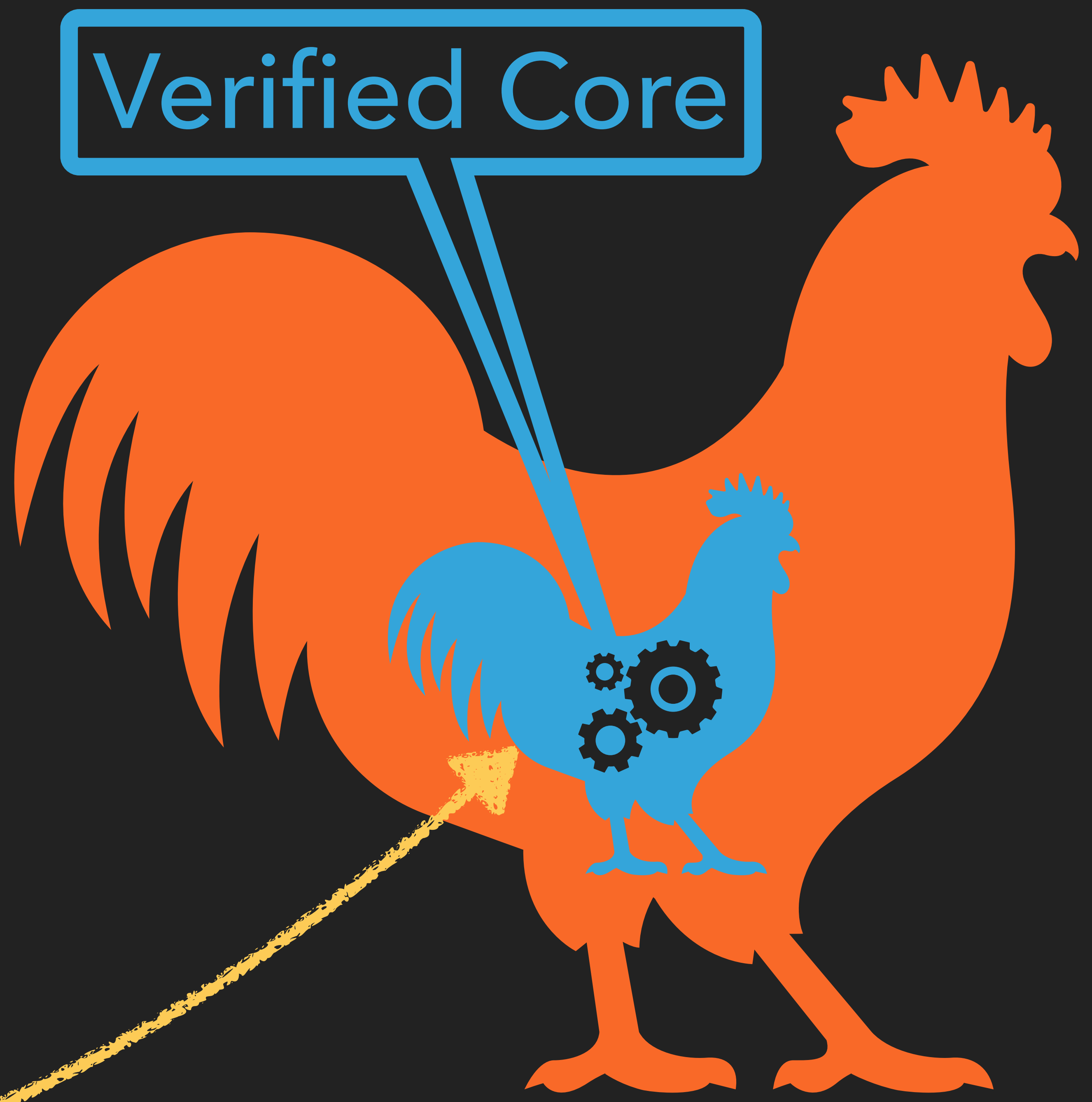
Verified Rocq

```
MetaRocq Check infer.
```

in

MetaRocq

in



Implemented Rocq
=
Ideal Rocq

Verified ℰ + Verified Extraction

```
MetaRocq Compile infer.
```

In the works

- Integration of Sort Polymorphism and Elimination Constraints (J. Rosain)
- Integration of an efficient verified algorithm for universe checking (M. Sozeau)
- Meta-Theory of eta-conversion, definitional proof-irrelevance, rewrite rules, typed equality, … (Y. Leray, …)
- Induction principles for nested types (T. Lamiaux, …)

Future directions

- Adding explicit existential variables for programming tactics / elaborations. ~ Lean's MetaM functionality.
- Extending the support for (verified) Meta-Programming (M. Bouverot-Dupuis, Y. Forster).

Part II
Formalization Challenges

On-demand separation of computational content

- Explicit `squash : Type -> Prop` (noted ‖ T ‖) instead of everything in Prop by default.
- Allows well-founded induction on derivations (or their size)
- Explicits the non-computational/computational distinction in statements, e.g. conversion:

  conv : forall Γ T U, ‖ isType Γ T ‖ -> ‖ isType Γ U ‖ ->
  ‖ Γ ⊢ T = U ‖ + ‖ ~ Γ ⊢ T = U ‖

- After erasure, a boolean is returned and no typing derivations are taken.

Essential use of dependent elimination

```
Lemma invert_type_mkApps_ind {cf:checker_flags} Σ Γ ind u args T mdecl idecl :
  wf Σ.1 →
  declared_inductive Σ.1 ind mdecl idecl →
  Σ ;;; Γ |- mkApps (tInd ind u) args : T →
  (typing_spine Σ Γ (subst_instance u (ind_type idecl)) args T)
  * consistent_instance_ext Σ (ind_universes mdecl) u.
Proof.
  intros wfΣ decli.
  intros H; dependent induction H; solve_discr.
```

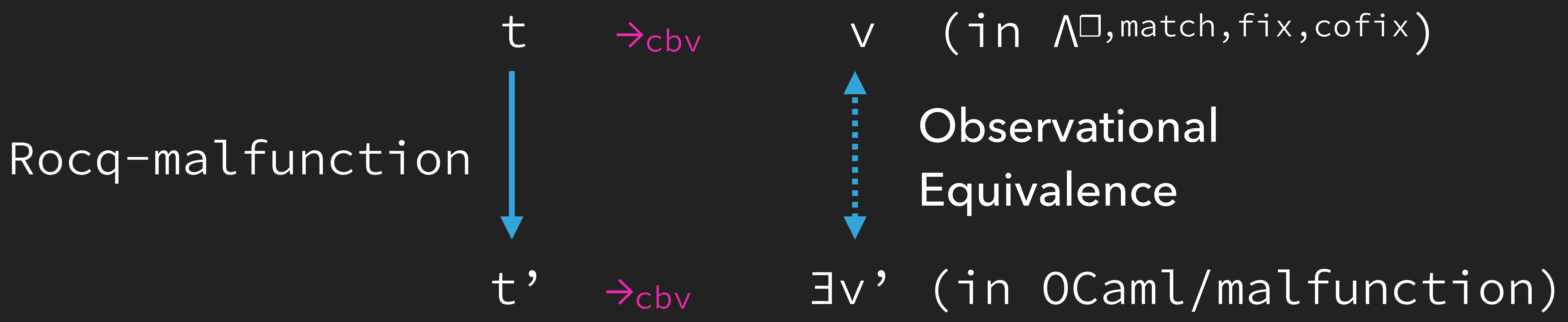
"Green slime" in hypotheses is common !

Essential use of (dependent) views

```
Equations? reduce_to_sort (Γ : context) (t : term)
  (h : ∀ Σ (wfΣ : abstract_env_ext_rel X Σ), welltyped Σ Γ t)
  : typing_result_comp (∑ u, ∀ Σ (wfΣ : abstract_env_ext_rel X Σ), ∥ Σ ;;; Γ ⊢ t ⇝ tSort u ∥) :=
  reduce_to_sort Γ t h with view_sortc t := {
    | view_sort_sort s ⇒ Checked_comp (s; _);
    | view_sort_other t _ with inspect (hnf Γ t h) :=
      | exist hnft eq with view_sortc hnft := {
        | view_sort_sort s ⇒ Checked_comp (s; _);
        | view_sort_other hnft r ⇒ TypeError_comp (NotASort hnft) _
      }
  }.
```

```
Inductive view_sort : term → Type :=
| view_sort_sort s : view_sort (tSort s)
| view_sort_other t : ~ isSort t → view_sort t.
```

Feasibility of formalization



- ~ 10 compiler passes to formalize
- Slight variants of the AST are used
- For feasibility => configurable AST and well-formedness
- Custom induction principle building in well-formedness

Configurable ASTs and relations

- For lamba-box: only one AST definition and evaluation relation
- Well-formedness and evaluation are configured by a set of flags activating or deactivating specific constructors or rules.
- Advantage: generic lemmas for all possible combinations, makes apparent the pre/post-conditions of each phase.

Avoid duplication!

E.g. when transforming constructor applications to blocks, we disallow generic application to have a constructor at the head, disable the application congruence rule and enable a specific constructor congruence rule.

Custom Induction Principles

- Idea: combine an inductive property on terms with the induction principle for terms themselves.
- Equivalent to working with a subset type {x : term | P x} without the currying/uncurrying administrative overhead.
- Does the boilerplate invariant passing once and for all.
- Example: evaluation of well formed terms, without having to invoke preservation of wellformedness at each step (in 10 proofs)
- Related to Ornaments (McBride et al)

Nested inductives for reuse

- Many specifications and proofs rely on lists of data being synchronized, making essential use of nested inductive types.

All2 (fun b bty => |- b : bty) branches branches_types

- Large reusable library around the use of All / All2 / Alli / All_fold predicates on multiple lists, and their dependent versions, e.g:

```
Inductive All2i {A B : Type} (R : nat → A → B → Type) (n : nat)
  : list A → list B → Type :=
| All2i_nil : All2i R n [] []
| All2i_cons x y l r : R n x y → All2i R (S n) l r → All2i R n (x :: l) (y :: r).
```

- Different from `In` or big operator algebra (AFAICT)

Nested inductives are crucial but badly supported

- Derivation of nested elimination principles is manual in Rocq, only a restrictive subset of nesting is supported by Lean. Well supported by BNFs in Isabelle
- **News** We have a generic methodology applicable to both Rocq and Lean to generate **user-friendly** eliminators based on “sparse” parametricity (T. Lamiaux, Y. Forster, M. Sozeau, N. Tabareau). Defined in MetaRocq, WIP plugin for Rocq

Some lessons learned

- Avoiding duplication and smart proof engineering is essential for feasibility of these proofs. E.g. establishing powerful elimination principles.
- Modularity and genericity are key to avoid duplication, e.g. through the use of nested inductive types and polymorphic predicates

Related Work

- Coq in Coq (Barras) - normalization, idealized calculus
- MLTT in Agda formalisations (Abel et al) - focus on normalization/consistency, NbE algorithm, erasure
- Martin Löf à la Coq (Adjedj et al) - variant of Abel et al.
- Lean4Lean (Carneiro)
- CakeML / Candle (Myreen et al)

Going further

- See metarocq.github.io for documentation, papers and examples
- Part of the Rocq Platform

MetaRocq

Verified