HoTTEST Seminar

The MetaCoq Project

Matthieu **Sozeau**

Inria & LS2N, University of Nantes

joint work with

Abhishek **Anand**
Bedrock Systems, Inc

Danil **Annenkov**
University of Copenhagen

Simon **Boulier**
University of Nantes

Cyril **Cohen**
Inria

Yannick **Forster**
University of Saarland

Meven **Lennon-Bertrand**
University of Nantes

Gregory **Malecha**
Bedrock Systems, Inc

Jakob **Botsch Nielsen**
University of Copenhagen

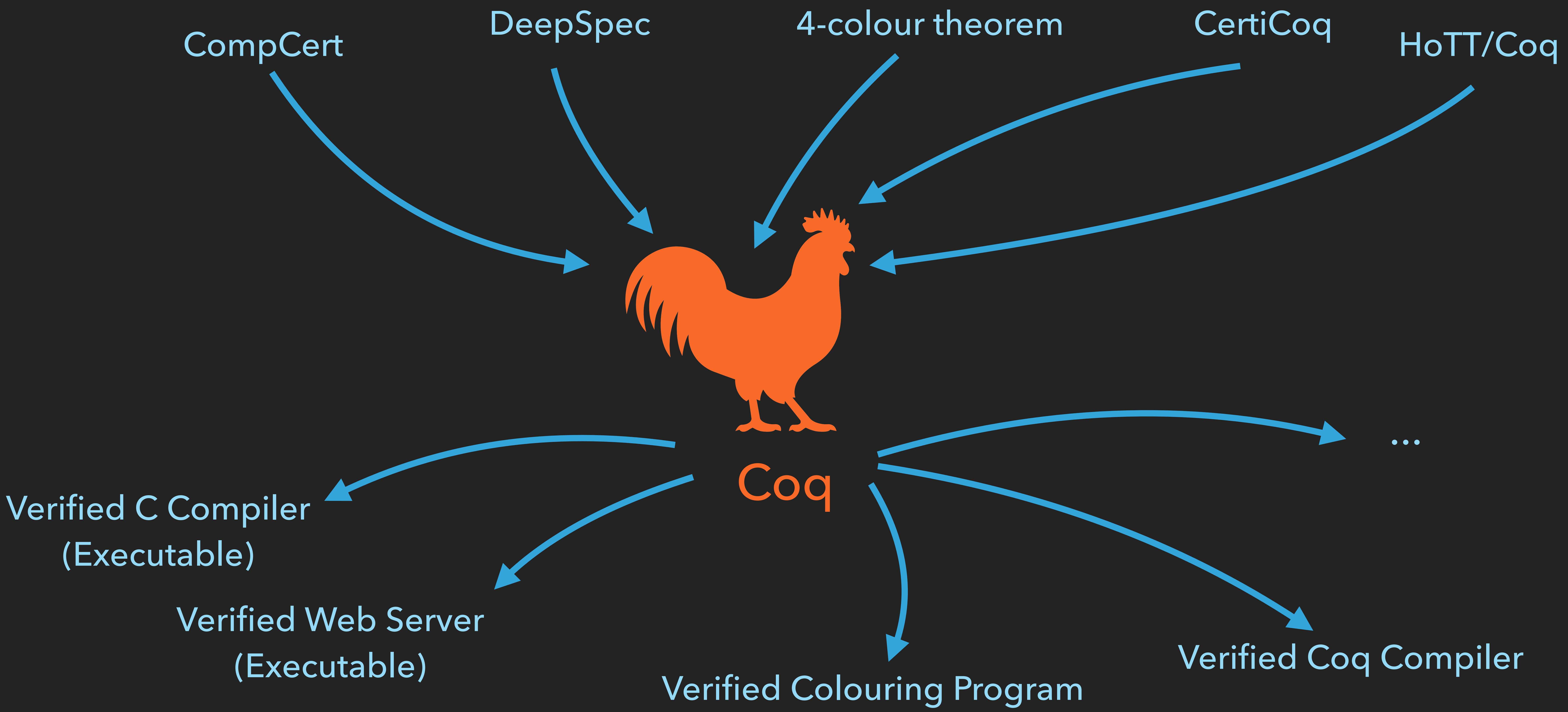
Nicolas **Tabareau**
Inria & LS2N

Théo **Winterhalter**
Inria & LS2N

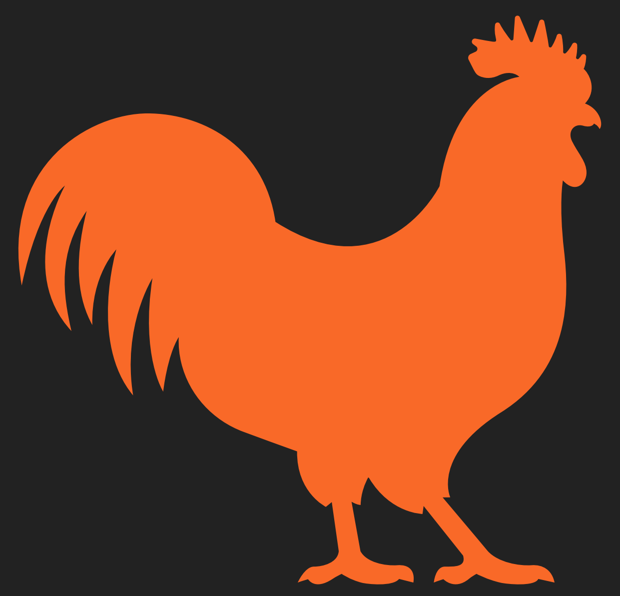
The MetaCoq Team

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

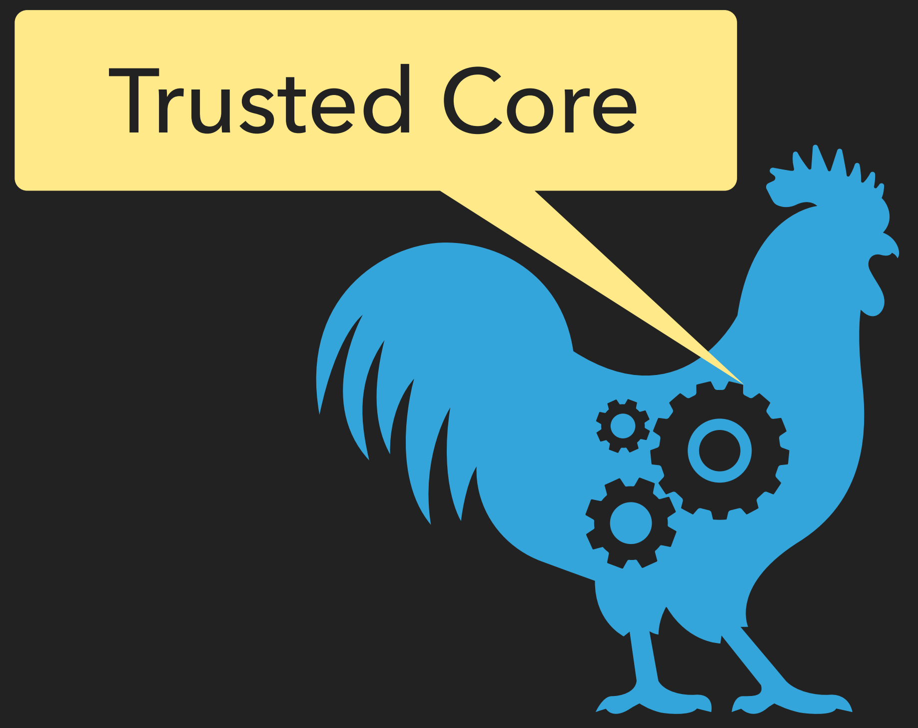
Motivation


CompCert
DeepSpec
4-colour theorem
CertiCoq
HoTT/Coq
Coq
Verified C Compiler
(Executable)
Verified Web Server
(Executable)
Verified Colouring Program
Verified Coq Compiler
...

What do you trust?

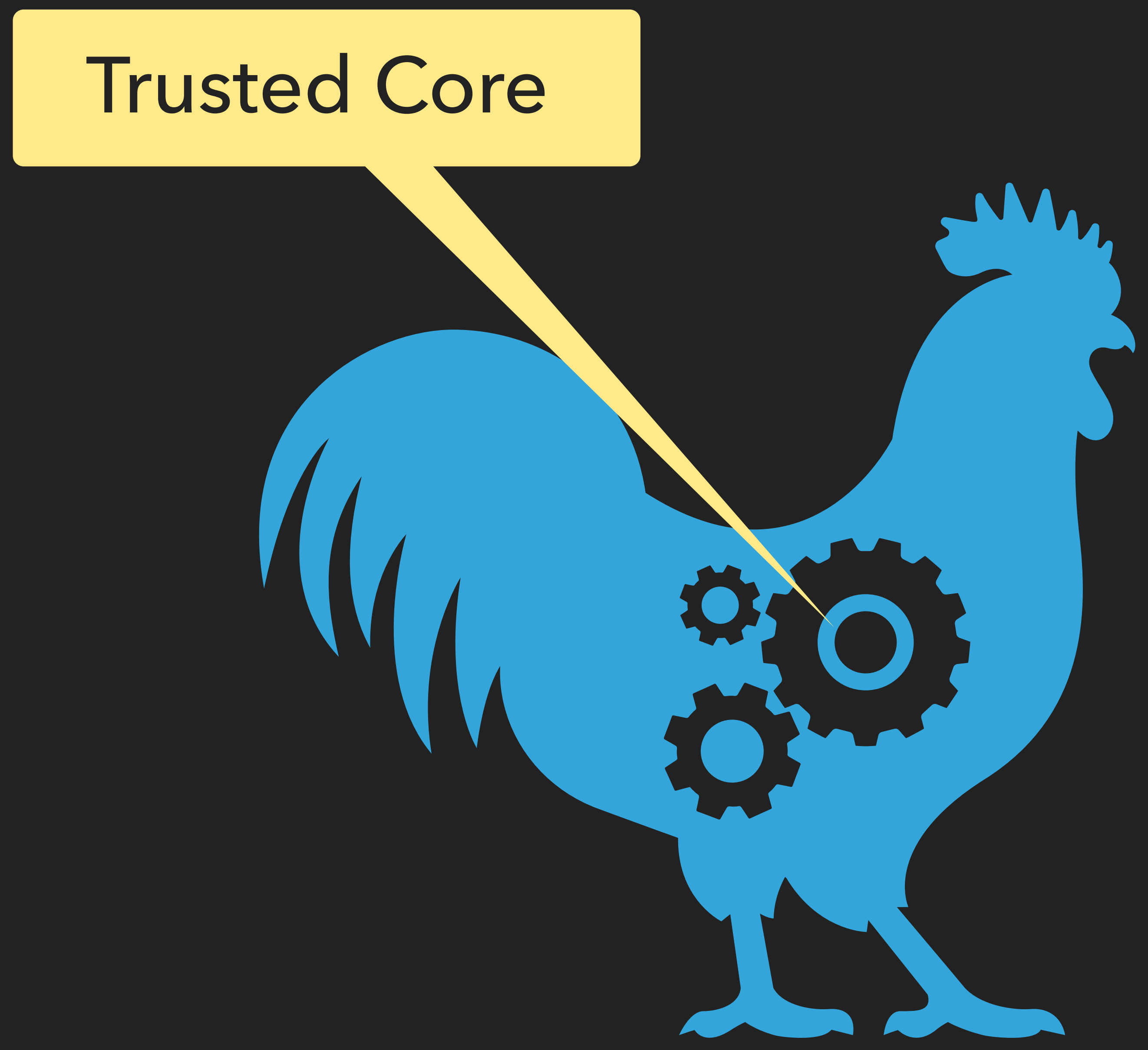


Ideal Coq



Implemented Coq

What do you trust?



Implemented Coq

Dependent Type Checker (18kLoC, 30+ years)

- Inductive Families w/ Guard Checking
- Universe Cumulativity and Polymorphism
- ML-style Module System
- KAM, VM and Native Conversion Checkers
- OCaml's Compiler and Runtime

The Reality

Unspecified

Ideal Coq

Buggy

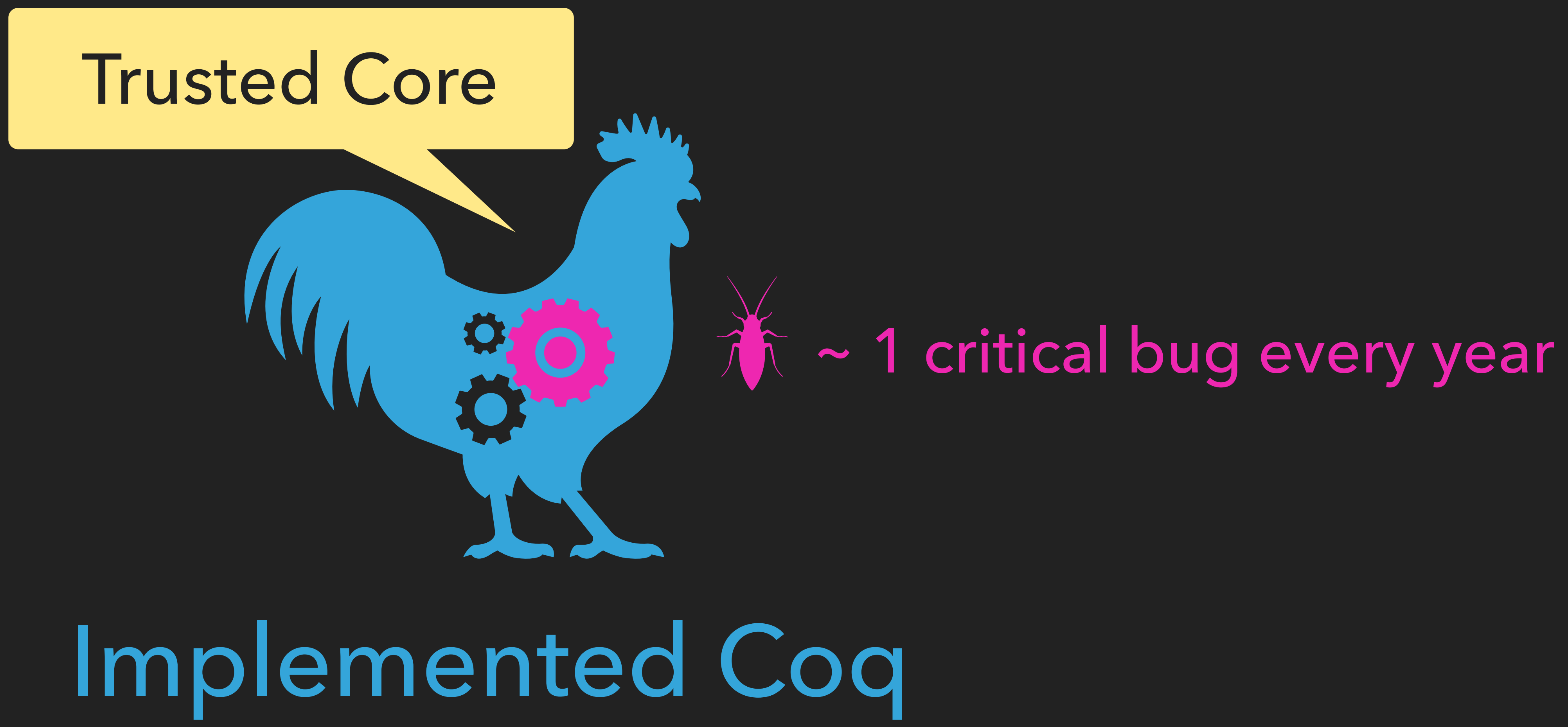
Implemented Coq

The Reality

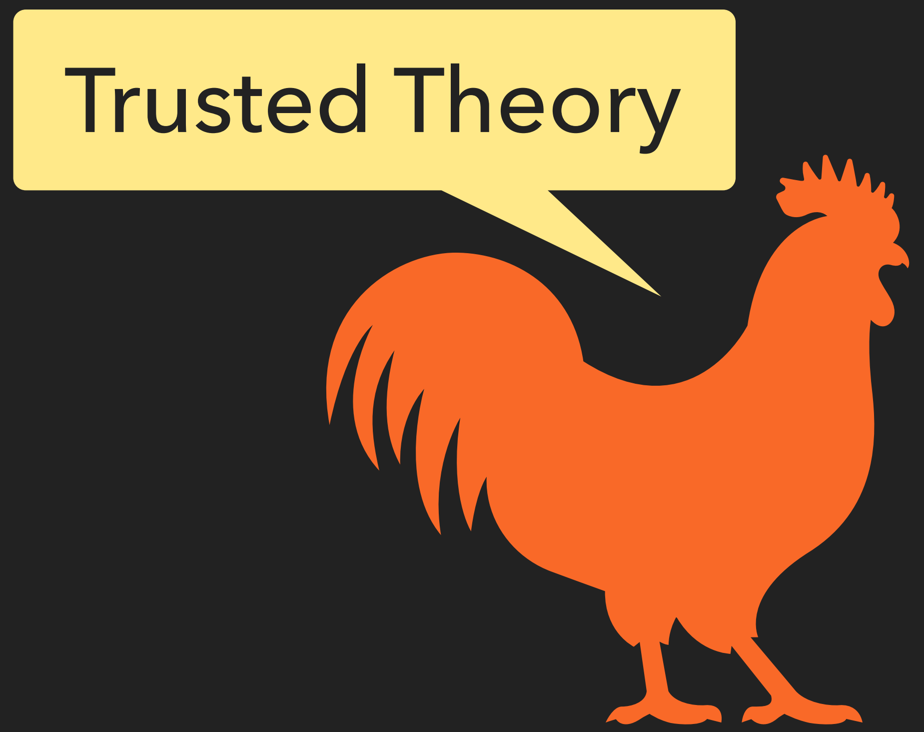
- Reference Manual is semi-formal and partial
- “One feature = n paper/PhD” where `n : fin 5`
  e.g. modules, universes, eta-conversion, guard condition, SProp….
- “Discrepancies” with the OCaml implementation
- Combination of features not worked-out in detail.
  E.g. cumulative inductive types + let-bindings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

The Reality

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

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


To add: #7723 (vm_compute universe polymorphism), #7695 (modules an

Typing constructions

  component: "match"
  summary: substitution missing in the body of a let
  introduced: ?
  impacted released versions: V8.3-V8.3pl2, V8.4-V8.4pl4
  impacted development branches: none
  impacted coqchk versions: ?
  fixed in: master/trunk/v8.5 (e583a79b5, 22 Nov 2015, Herbelin), v
  found by: Herbelin
```



Our Goal: Improving Trust



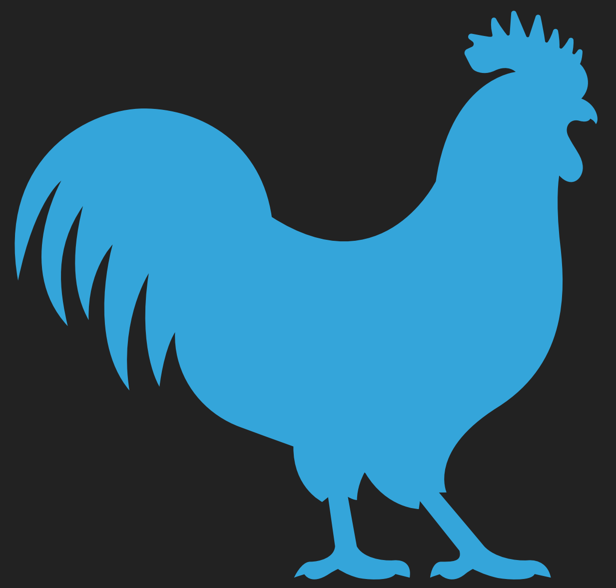
Ideal Coq

Implemented Coq

Coq in MetaCoq

Trusted Theory

Part I: Coq's Calculus PCUIC



Part II: Verified Coq

POPL'20

in

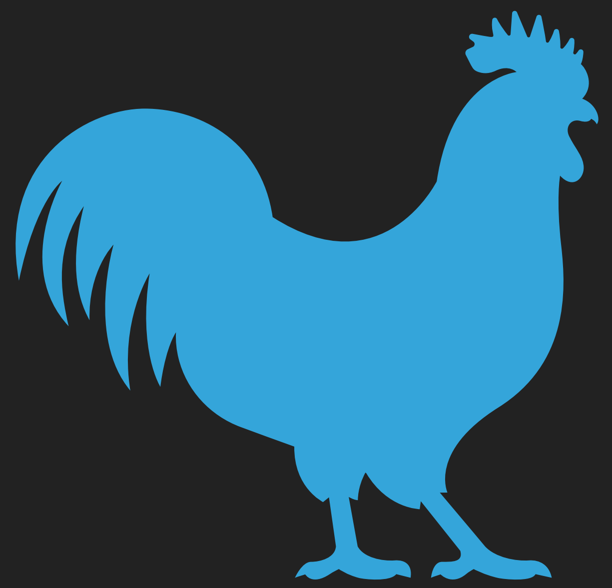
MetaCoq

Formalization of

Coq in Coq

ITP'19, JAR'20

in



Implemented Coq

MetaCoq in Practice

DEMO!

PCUIC

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

What we have...

```
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 := "Coq.Init.Datatypes.nat";
      inductive_ind := 0 |} [])
    (tProd (nNamed "m") (tInd {| ...
```

What we have...

```
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.
```

...and what we don't

~~`(fun x ⇒ f x) = f`~~ `(x ∉ FV(f))`

η-conversion (WIP)

~~`list nat : Set`~~
~~`list Type@{i} : Type@{i}`~~

« template » polymorphism

~~`Module M <: S. Definition t := nat. End M.`~~

module system

No existential or named variables (yet)

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 environment)

```
global_env_ext ::= (global_env × universes_decl)
```

(global environment with universes)

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

(local environment)

Meta-Theory

Judgments

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

One-step reduction and its reflexive transitive closure

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

α-equivalence + equality or cumulativity of universes

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

Conversion and cumulativity

$\Longleftrightarrow T \rightarrow^* T' \wedge U \rightarrow^* U' \wedge T' \leq_\alpha U'$

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

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 Coq
- 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
- Easier to lift to strengthening/exchange lemmas in the future (strengthening is not immediate here)

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 Coq, similar to Agda

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

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

- `lSet` always has level `0`, smaller than any other universe.
- Impredicative sorts are separate from the predicative hierarchy.

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)

Implementation

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

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

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 $\Rrightarrow$

$$\begin{tikzcd}
 & t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] & \\
v \arrow[dr, Rrightarrow] & & u \arrow[dl, Rrightarrow] \\
 & \exists\ t' &
\end{tikzcd}$$

"Squash" lemma

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

Takahashi's Trick

```
ρ : term -> term
```

An *optimal* one-step parallel reduction function.

$$t \Rrightarrow \rho(t)$$

The triangle property

$$
\begin{tikzcd}
t \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \\
\rho(t) & u \arrow[l, Rightarrow]
\end{tikzcd}
$$

The triangle property

$$\begin{tikzcd} & t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow] & \\ v \arrow[dr, Rrightarrow] & & u \arrow[dl, Rrightarrow] \\ & \rho(t) & \end{tikzcd}$$

Confluence

For a theory with definitions in contexts

$$\Sigma \vdash \Gamma,\ t \Rrightarrow \Delta,\ u$$

One-step parallel reduction, including reduction in contexts.

$$\frac{\Sigma \vdash \Gamma,\ x := t \Rrightarrow \Delta,\ x := t' \qquad \Sigma \vdash (\Gamma,\ x := t),\ b \Rrightarrow (\Delta,\ x := t'),\ b'}{\Sigma \vdash \Gamma,\ (\mathtt{let}\ x := t\ \mathtt{in}\ b) \Rrightarrow \Delta,\ (\mathtt{let}\ x := t'\ \mathtt{in}\ b')}$$

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

$$\begin{tikzcd}
 & \Gamma, t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow] & \\
\Gamma', u \arrow[dr, Rrightarrow] & & \Gamma'', v \arrow[dl, Rrightarrow] \\
 & \rho ctx\ \Gamma,\ \rho\ (\rho ctx\ \Gamma)\ t &
\end{tikzcd}$$

Principality and changing equals for equals

```
Definition principality {Σ Γ t} : (welltyped Σ Γ t : Prop) →
  ∑ (P : term), Σ ; Γ ⊢ t : P × principal_type Σ Γ t P
```

$$\frac{\Sigma \,;\, \Gamma \vdash t : T \qquad \Sigma \,;\, \Gamma \vdash u : U \qquad \Sigma \vdash u \leq_{\alpha\_noind} t}{\Sigma \,;\, \Gamma \vdash u : T}$$

Informally: (well-typed) smaller terms have more types than larger ones.

Justifies the `change` tactic up-to cumulativity (excluding inductive type cumulativity).

Cumulativity and Prop/SProp

$$\Sigma \; ; \; \Gamma \vdash T \sim U$$

Conversion identifying all predicative universes (hence larger than cumulativity).

$$\frac{\Sigma \; ; \; \Gamma \vdash t : T \qquad \Sigma \; ; \; \Gamma \vdash u : U \qquad \Sigma \vdash u \leq_\alpha t}{\Sigma \; ; \; \Gamma \vdash T \sim U}$$

Informally: for two well-typed terms, if they are syntactically equal up-to cumulativity of inductive types, then they live in the same hierarchy (`Prop`, `SProp` or `Type`)

Required for **erasure correctness**

Alternative to Letouzey's restricted system when **Prop** $\not\leq$ **Type**

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 Coq implementation of the guard/productivity checkers

Trusted Theory Base

Assumptions

```
Axiom normalisation :
  ∀ Σ Γ t, welltyped Σ Γ t → Acc (cored (fst Σ) Γ) t.
```

- Strong Normalization
  Not provable thanks to Gödel
- Consistency and canonicity follow easily.
- Used exclusively for the termination proof of the conversion test
- Could be inherited by preservation of normalisation from a stronger system with a model

Verifying Type-Checking

Conversion

Objective

Input

`u : A` `v : B`

Output

`(u ≡ v) + (u ≢ 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

$$u : A \xrightarrow{\text{whnf}} u' \qquad v' \xleftarrow{\text{whnf}} v : B$$

Conversion

Algorithm

$$u : A \xrightarrow{\text{whnf}} u' \overset{?}{\equiv} v' \xleftarrow{\text{whnf}} v : B$$

Conversion

Algorithm

$u : A \xrightarrow{\text{whnf}} u'$ $\qquad$ $v : B \xrightarrow{\text{whnf}} v'$

$\texttt{match}\ u' \overset{?}{\equiv} v'\ \texttt{with}$

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

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

Conversion

Completeness

$u : A \xrightarrow{\text{whnf}} u'$ $\qquad$ $v : B \xrightarrow{\text{whnf}} v'$

```
match u', v' with
```

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

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

Conversion

Completeness

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

Conversion

Completeness

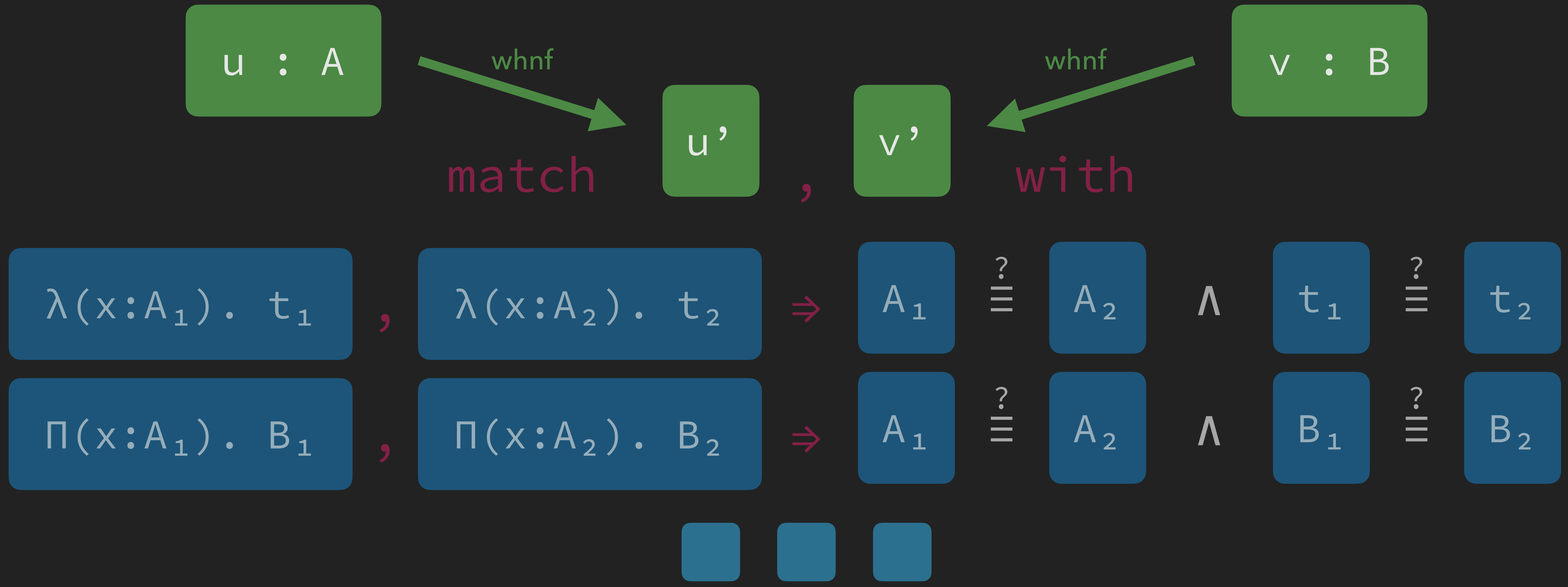
$\Pi(x{:}A_1).\ B_1 \overset{?}{\equiv} \Pi(x{:}A_2).\ B_2 \Rightarrow A_1 \not\equiv A_2$

we conclude

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

using inversion lemmata and confluence

Conversion



u : A
whnf
u'
match
,
v'
with
whnf
v : B
λ(x:A₁). t₁
,
λ(x:A₂). t₂
⇒
A₁ ≡? A₂ ∧ t₁ ≡? t₂
Π(x:A₁). B₁
,
Π(x:A₂). B₂
⇒
A₁ ≡? A₂ ∧ B₁ ≡? B₂

Weak head reduction

Objective

Input

u

`term`

Output

v

`term`

Weak head reduction

Objective

Input	Output	
`u`	`v`	`u -> v`
`term`	`term`	`Prop`

Weak head reduction

Objective

Input

u

term

Output

v

term

u -> v

Prop

```
weak_head_reduce : ∀ (u : term), ∑ (v : term), u -> v
```

Weak head reduction

Example

Input u

Output v u -> v

```
Definition foo := λ(x:nat). x.
```

```
foo 0
```

Weak head reduction

Example

Input u

Output v u -> v

```
Definition foo := λ(x:nat). x.
```

foo 0

foo $\longrightarrow$ λ(x:nat).x

Weak head reduction

Example

Input u

Output v u -> v

```
Definition foo := λ(x:nat). x.
```

```
λ(x:nat).x 0
```

```
foo ⟶ λ(x:nat).x
```

Weak head reduction

Example

Input u

Output v u -> v

```
Definition foo := λ(x:nat). x.
```

0

foo ⟶ λ(x:nat).x

Weak head reduction

Example

Input u

Output v u -> v

```
Definition foo := λ(x:nat). x.
```

0

foo 0 ⟶ (λ(x:nat).x) 0 ⟶ 0

Weak head reduction

Termination

Input

u

Output

v

u -> v

Weak head reduction

Termination

Input

u π_1

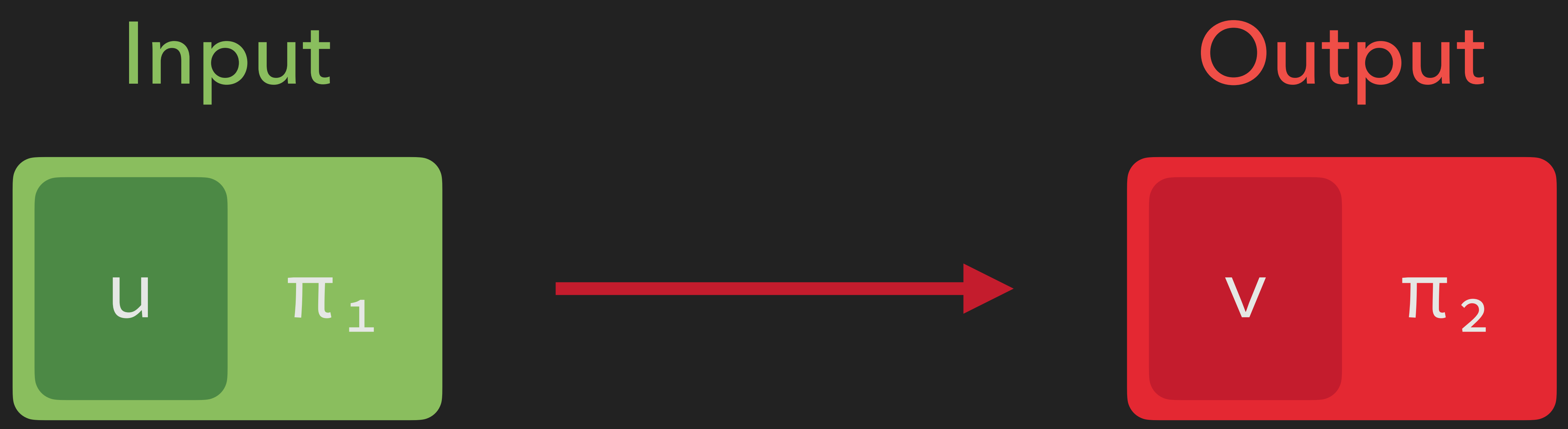
Output

v π_2

u -> v

Weak head reduction

Termination



Weak head reduction

Termination

`foo 0` `foo 0` `λ(x:nat).x 0` `0`

Weak head reduction

Termination

`foo 0` `foo 0` `λ(x:nat).x 0` `0`

`(λ(x:nat).x) 0` ⟶ `0`

Weak head reduction

Termination

```
foo 0 ⟶ (λ(x:nat).x) 0
```

```
foo 0    foo 0    λ(x:nat).x 0    0
```

```
(λ(x:nat).x) 0 ⟶ 0
```

Weak head reduction

Termination

`foo 0` ⟶ `(λ(x:nat).x) 0`

`foo 0` | `foo 0` | `λ(x:nat).x 0` | `0`

`foo 0 ⊐ foo`

`(λ(x:nat).x) 0` ⟶ `0`

Weak head reduction

Termination

```
foo 0 ⟶ (λ(x:nat).x) 0
```

`foo 0` | `foo 0` | `λ(x:nat).x 0` | `0`

```
foo 0 ⊐ foo          (λ(x:nat).x) 0 ⟶ 0
```

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

`foo 0` $\longrightarrow$ `(λ(x:nat).x) 0`

`foo 0` | `foo 0` | `λ(x:nat).x 0` | `0`

`foo 0 ⊐ foo`

`and foo 0 = foo 0`

`(λ(x:nat).x) 0` $\longrightarrow$ `0`

Lexicographic order of `->` and ⊏

Weak head reduction

Termination

`p.1`

1 Lexicographic order of `->` and $\sqsubset$

Weak head reduction

Termination

`p.1`

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

`p.1` `p.1`

but `p.1` $\neq$ `p`

1 Lexicographic order of `->` and $\sqsubset$

Weak head reduction

Termination

`p.1` > `p.1`

`and p.1 = p.1`

1

Lexicographic order of `->` and $\sqsubset$

Weak head reduction

Termination

```
fix f (n:nat). t end n
```

Lexicographic order of -> and ⊏

Weak head reduction

Termination

```
fix f (n:nat). t end n
```

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

```
fix f (n:nat). t end n
```

1 Lexicographic order of -> and $\sqsubset$

Weak head reduction

Termination

```
fix f (n:nat). t end n
```

▼

```
fix f (n:nat). t end n
```

~~Lexicographic order of $\to$ and $\sqsubset$~~

Weak head reduction

Termination

~~Lexicographic order of $\longrightarrow$ and $\sqsubset$~~

Weak head reduction

Termination

Lexicographic order of -> and an order on positions

Weak head reduction

Termination

Lexicographic order of -> and an order on positions

Weak head reduction

Termination

$u\ \pi_1 \ \blacktriangleright\ v\ \pi_2$

2 Lexicographic order of -> and an order on positions

Weak head reduction

Termination

$u\ \pi_1 \blacktriangleright v\ \pi_2$

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

Lexicographic order of -> and an order on positions

Weak head reduction

Termination

$$u\ \pi_1 \blacktriangleright v\ \pi_2$$

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

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

Type Checking

Weak head reduction

Conversion

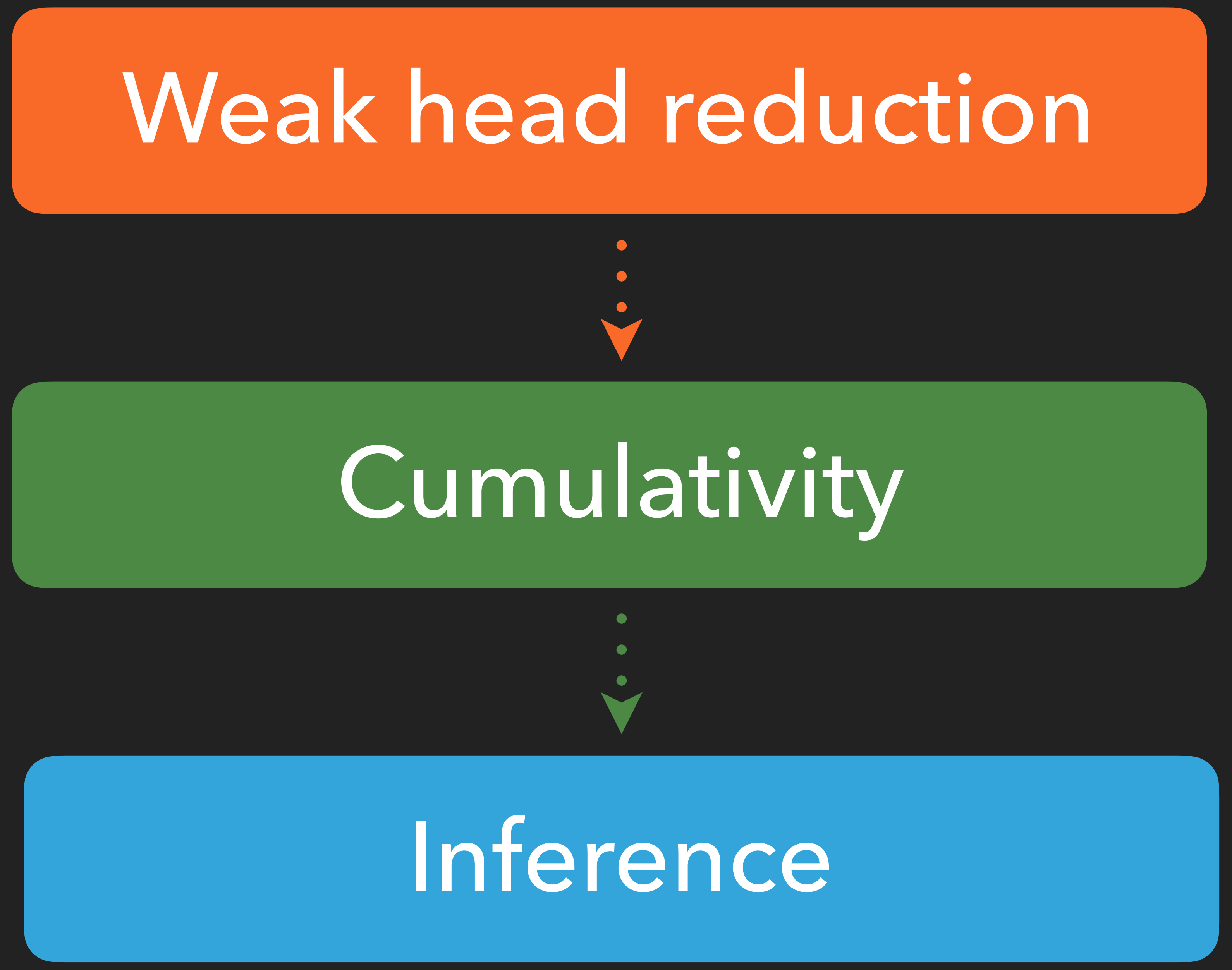
Type Checking

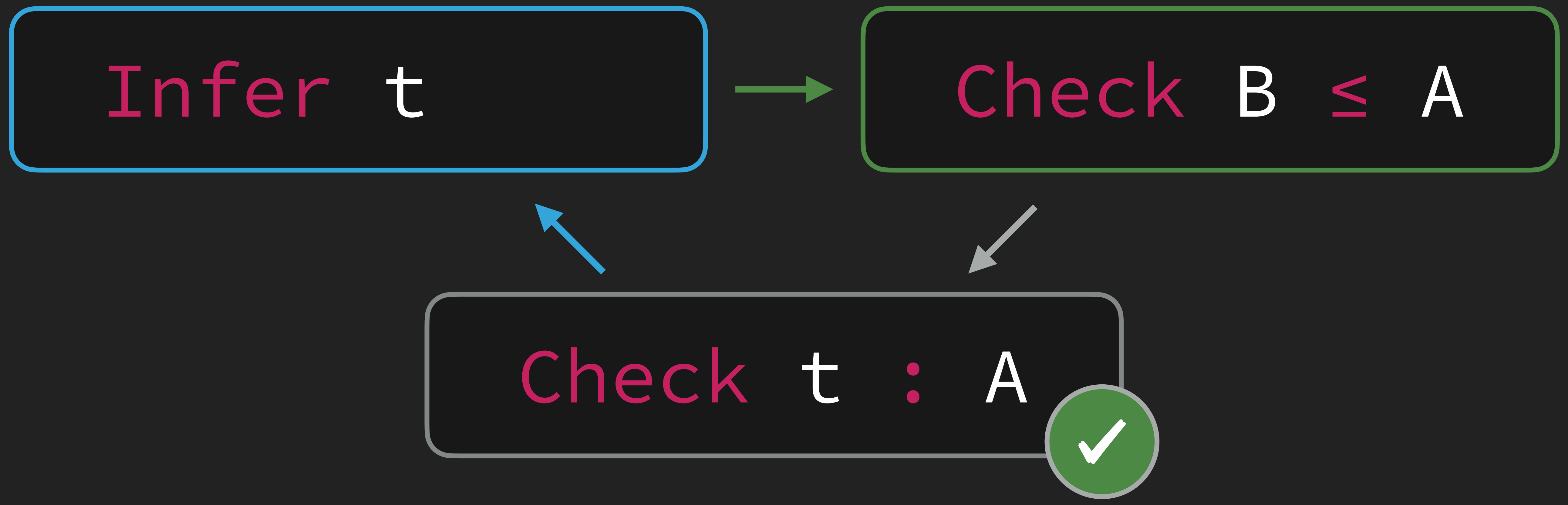
Weak head reduction

Cumulativity

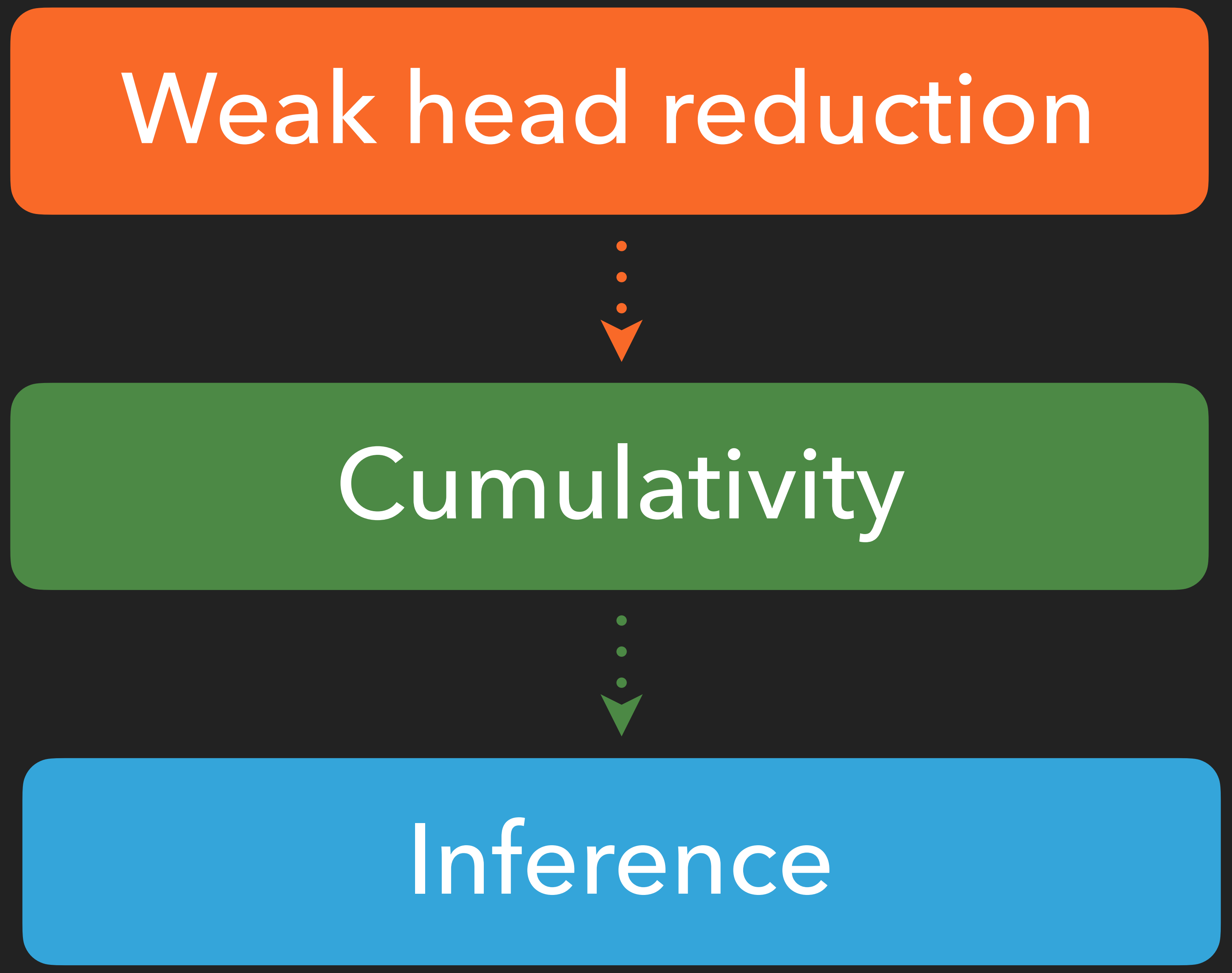
Inference

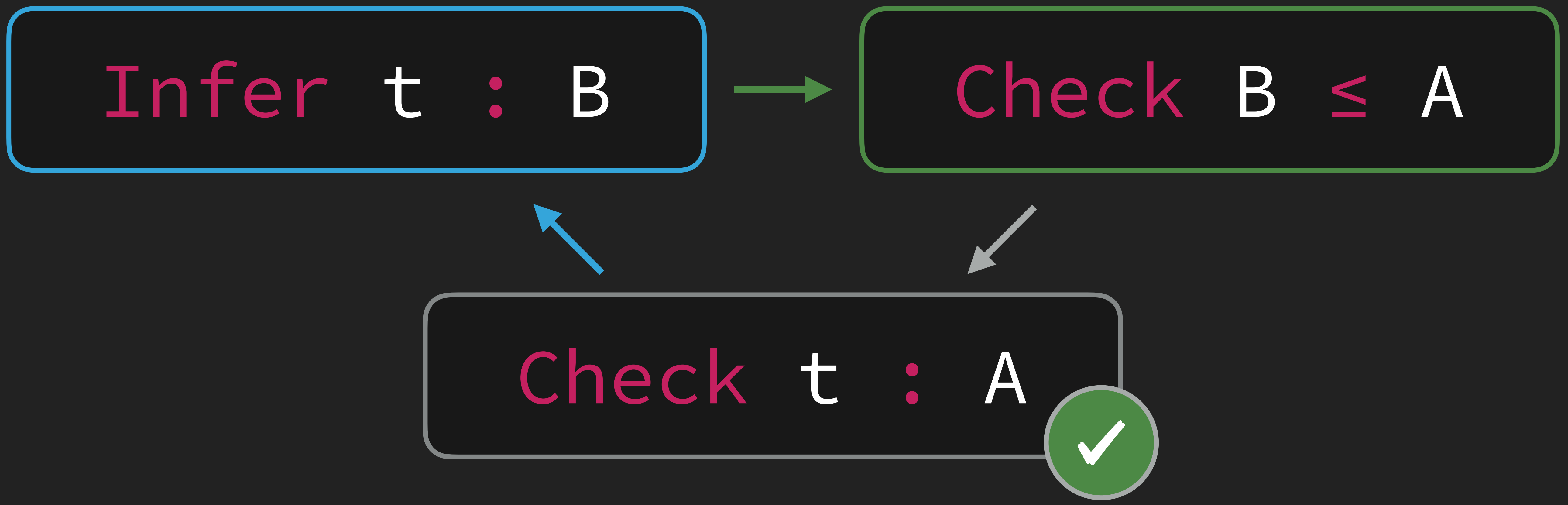
Type Checking





Type Checking






```
MetaCoq Check foo.
```

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

Confusion:

```
(fun (y : A) (e : eq@{i} A x y) => P y e)
                    ≢
(fun (y : A’) (e : eq@{j} 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 fails in Coq (Coq issue [#13495])
- Quick & dirty fix requires **strengthening**, not provable by induction

```
Confusion:
  (fun (y : A) (e : eq@{i} A t y) => P y e)
                      ≢
  (fun (y : A’) (e : eq@{j} A’ t y) => P y e)
```

A little success story

$$\frac{\Sigma ; \Gamma \vdash A : \mathtt{Type} \qquad \Sigma ; \Gamma \vdash t : A \qquad \Sigma ; \Gamma , x : A , e : \mathtt{eq@\{i\}}\ A\ t\ x \vdash P : \mathtt{Type} \qquad \Sigma ; \Gamma \vdash p : \mathtt{eq@\{i\}}\ A\ t\ u}{\Sigma ; \Gamma \vdash \mathtt{match}\ p\ \mathtt{as}\ e\ \mathtt{in}\ \mathtt{eq@\{i\}}\ A\ t\ x\ \mathtt{return}\ P\ \mathtt{with}\ \mid \mathtt{eq\_refl} \Rightarrow b\ \mathtt{end} : P\ u\ p}$$

- **Idea**: let case carry only the parameters and instance, build the derivable predicate and branches contexts on the fly.
- Completeness holds, reflects the high-level user syntax more faithfully. It’s win/win!

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 **transitivtity** of conversion
- **Strengthening** follows directly: justifies `clear`

ITP'21 Bidirectional type checking for CIC

Linking the spec to a typed equality

```
Σ ; Γ ⊢ t, u : T and
Σ ; Γ ⊢ t = u (defined by reduction)

  vs

Σ ; Γ ⊢ t = u : T (defined by a judgement)
```

- Untyped equality is shown to be an equivalence thanks to confluence of reduction (without relying on normalisation)
- Typed equality is generated from reduction rules, congruence rules and closed under reflexivity, symmetry and transitivity
- Typed equality implies untyped equality
- Needs SR in the typed equality system to show the converse

Decidability of conversion for Type Theory in Type Theory

Abel et al. (POPL'18)

- SR requires both substitutivity and injectivity of type constructors.

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

	Substitutivity	Injectivity of type constructors
Algorithmic System	Requires Normalization	Direct (whnfs comparisons)
Declarative System	Direct	Logical Relation Argument

Why does SR break

SR breaks down because:

$$\frac{\Sigma \; ; \; \Gamma \vdash (\text{fun } x : A \Rightarrow t) : \Pi\, A'\, B' \qquad \Sigma \; ; \; \Gamma \vdash u : A'}{\Sigma \; ; \; \Gamma \vdash (\text{fun } x : A \Rightarrow t)\; b : B'[t/x]}$$

To show b[t/x] : B'[t/x] requires to invert the first derivation:

$$\Sigma \; ; \; \Gamma , x : A \vdash t : B \quad /\backslash$$
$$\Sigma \; ; \; \Gamma \vdash \Pi\, A\, B = \Pi\, A'\, B' : s$$

and use injectivity of Π-types to conclude A = A' and B = B'. But injectivity of type constructors in turn usually requires a logical relation argument.

Bidirectionality to the rescue?

With bidirectional typing rules:

$$\frac{\Sigma \;;\; \Gamma \vdash (\mathtt{fun}\ x : A \Rightarrow t) \Uparrow T \qquad T \to_{whnf} \Pi\ A'\ B' \qquad \Sigma \;;\; \Gamma \vdash u \Downarrow A'}{\Sigma \;;\; \Gamma \vdash (\mathtt{fun}\ x : A \Rightarrow t)\ b \Uparrow B'[t/x]}$$

To show b[t/x] : B'[t/x] requires to invert the first derivation:

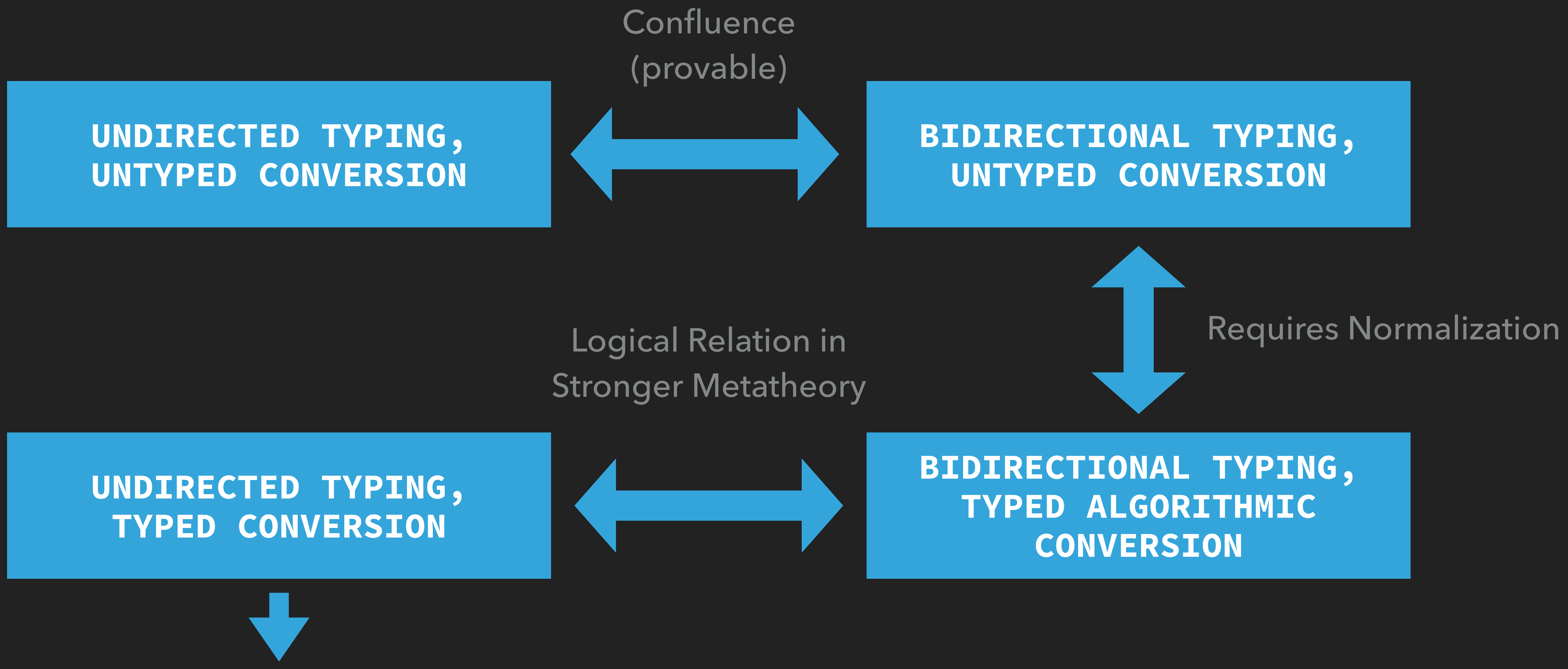
$$\Sigma \;;\; \Gamma , x : A \vdash t \Uparrow B \quad /\backslash \ T = \Pi\ A\ B$$

$$\Pi\ A\ B \to_{whnf} \Pi\ A'\ B' \text{ directly implies}$$

$$A = A' \text{ and } B = B'$$

Substitutivity allows to conclude, but hard needs SN.

Towards standard models



Conjecture: lifts to CIC

Open problem: impredicativity in the internal model

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:

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

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

```
Ɛ (coerce) ~   coerce x y := (fun p ⇒ p)
```

```
Ɛ (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

$$
\begin{tikzcd}
t \arrow[r, "\to_{cbv}"] \arrow[d, "\mathcal{E}"'] & v \arrow[d, dotted, leftrightarrow, "\text{Observational Equivalence}"] \\
t' \arrow[r, "\to_{cbv}"] & \exists v'
\end{tikzcd}
$$

With Canonicity and SN:

$$
\begin{aligned}
& \vdash t : nat \\
\Rightarrow\ & \vdash t \to n : nat \qquad (n \in \mathbb{N}) \\
\Rightarrow\ & t \to_{cbv} n : nat \\
\Rightarrow\ & \mathcal{E}(t) \to_{cbv} n
\end{aligned}
$$

Erasure Correctness

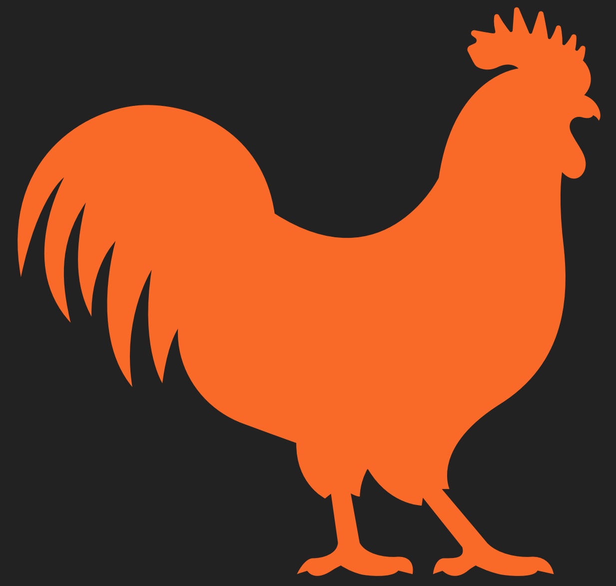
First define a non-deterministic erasure relation, then define:

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

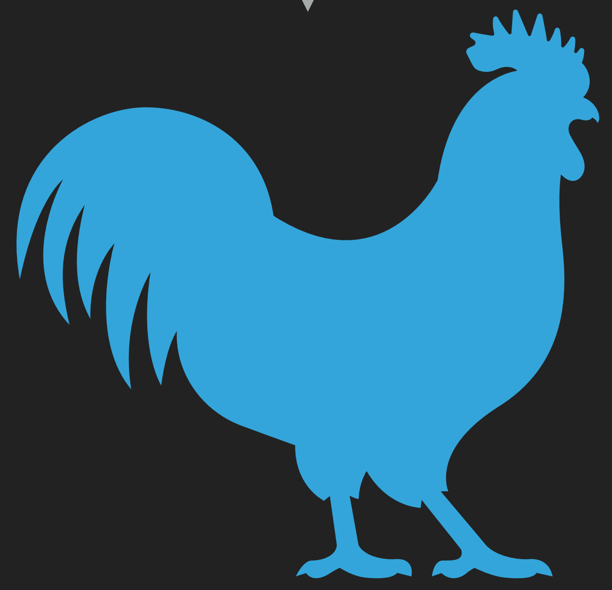
Finally show that ℰ's graph is in the erasure relation. Two additional optimizations:

- 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.

Summary



Ideal Coq

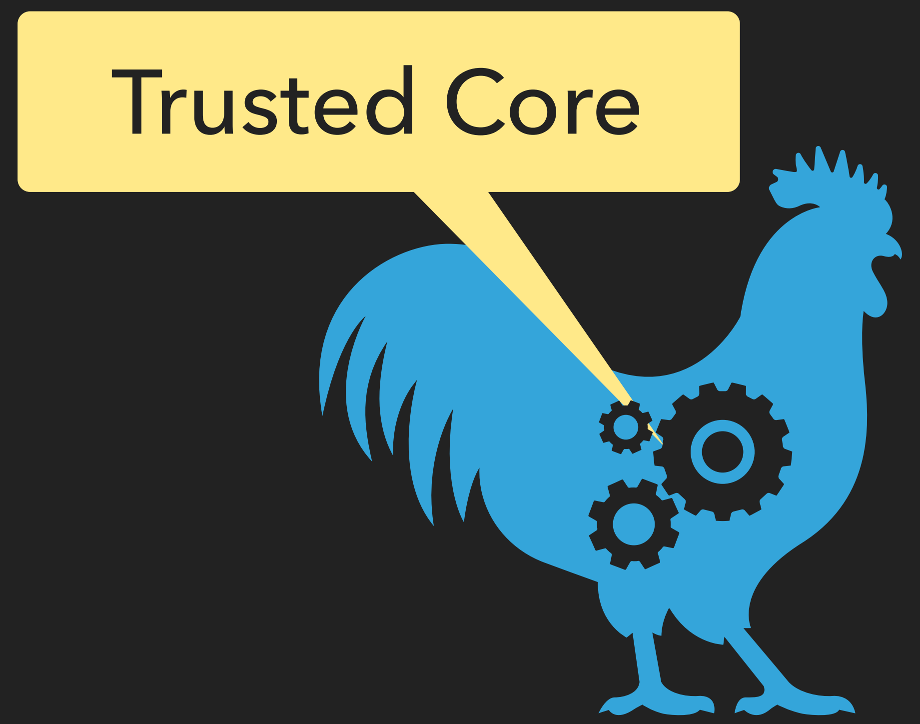


Verified Coq

in

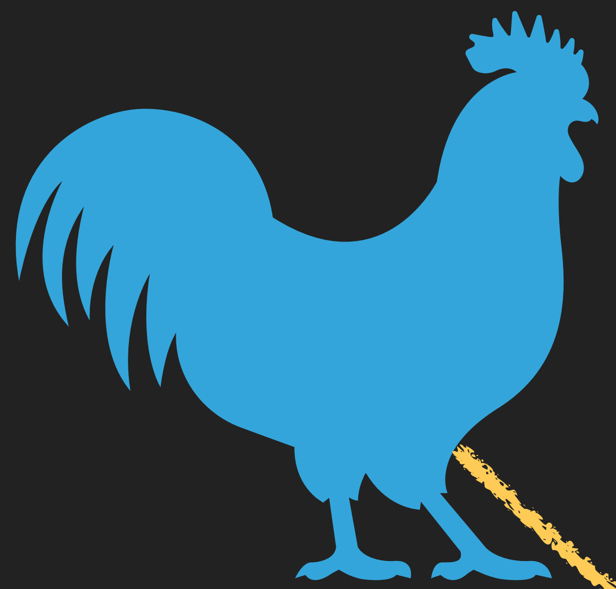
MetaCoq

in



Implemented Coq

Summary



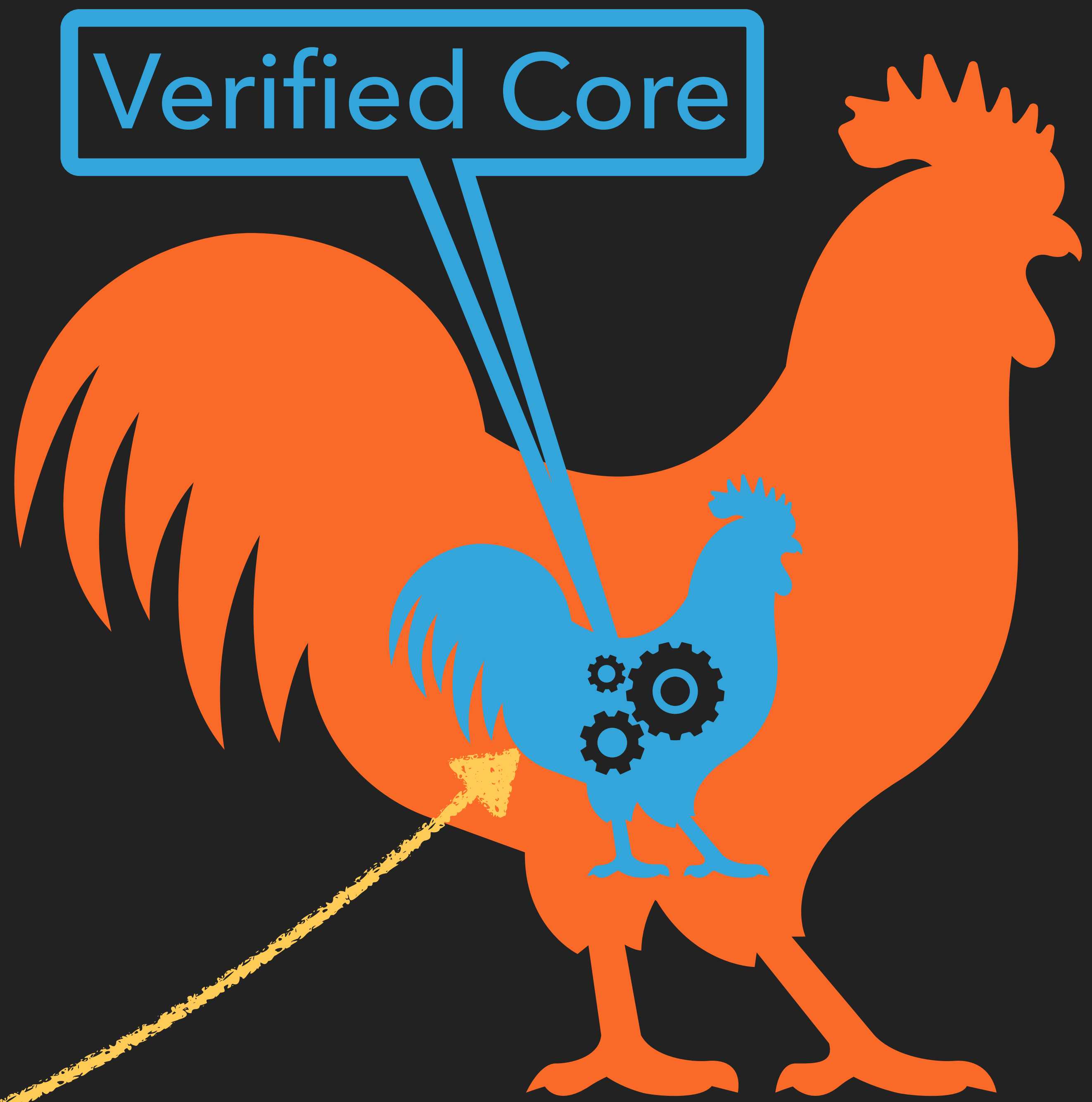
Verified Coq

```
MetaCoq Check vrev.
```

in

MetaCoq

in



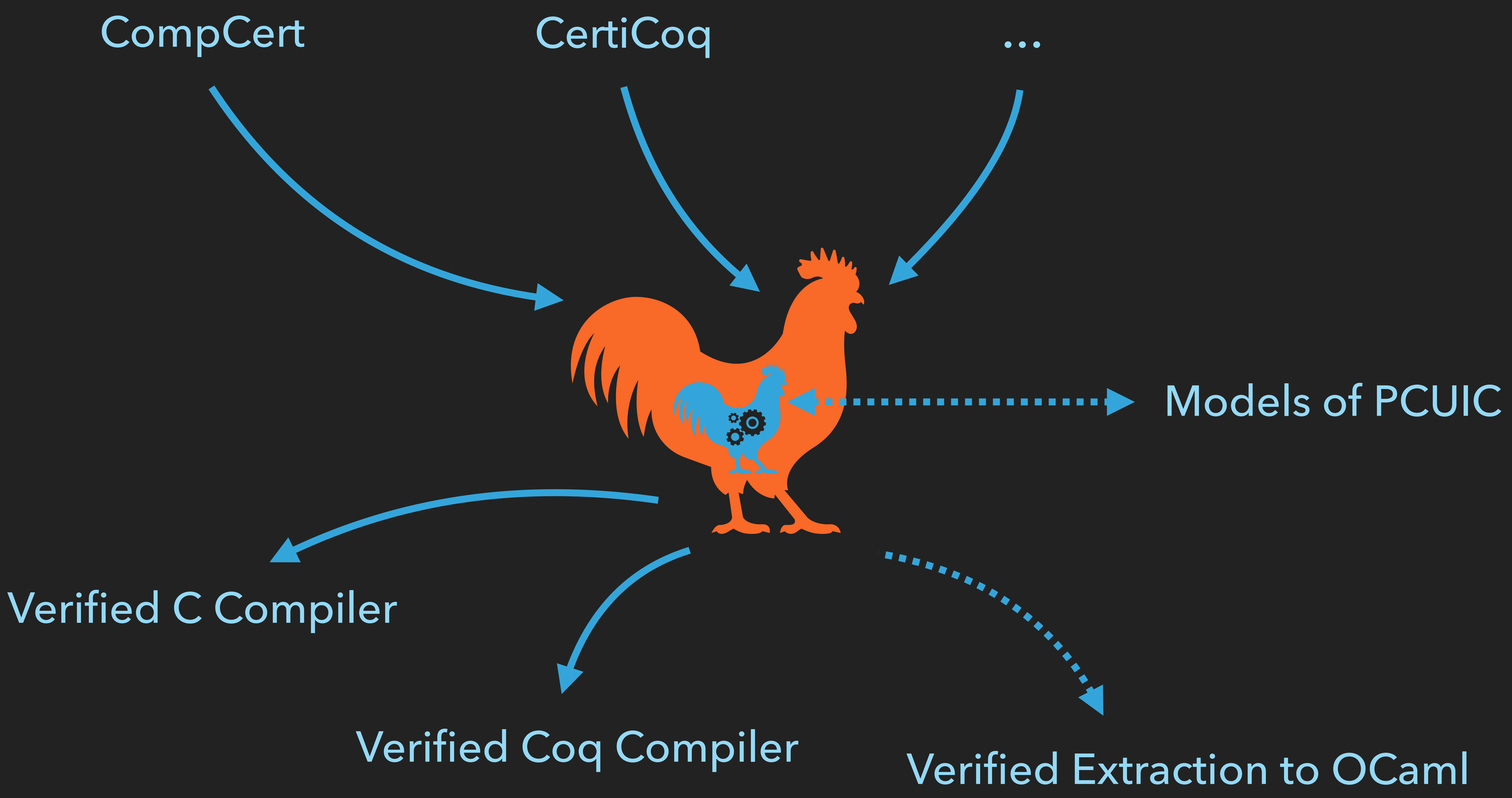
Implemented Coq
=
Ideal Coq

Verified ℰ

```
MetaCoq Erase vrev.
```

Spec:	30kLoC
Proofs:	60kLoC
Comments:	10kLoC

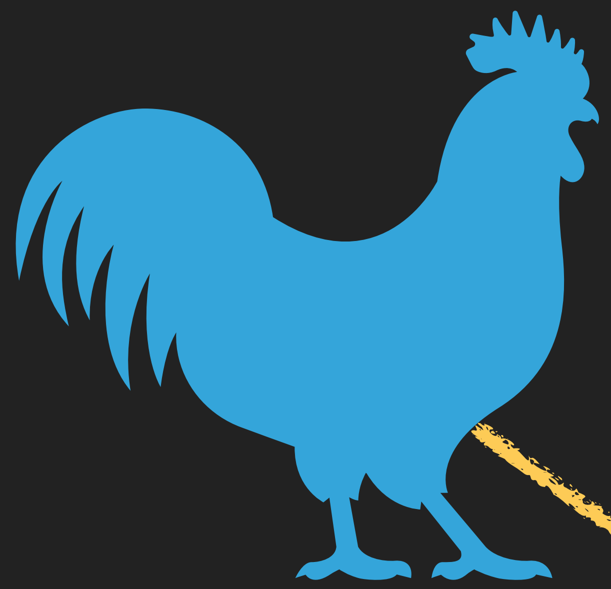
Perspectives



Ongoing and future work

- Verified Translations / Syntactic Models (e.g. presheaves, sheaves)
- Integration of **rewrite rules** (CEP #50, by Théo Winterhalter)
- Interoperability of erased code with OCaml
- Full meta-theory for the **SProp** sort and irrelevance checking
- **Eta-conversion** and contravariant subtyping (CEP #47)
- **Sort-polymorphism** generalising universe polymorphism to deal more uniformly with impredicative sorts and alternative hierarchies (exceptional type theory, setoid type theory, erasable sets…).

Conclusion



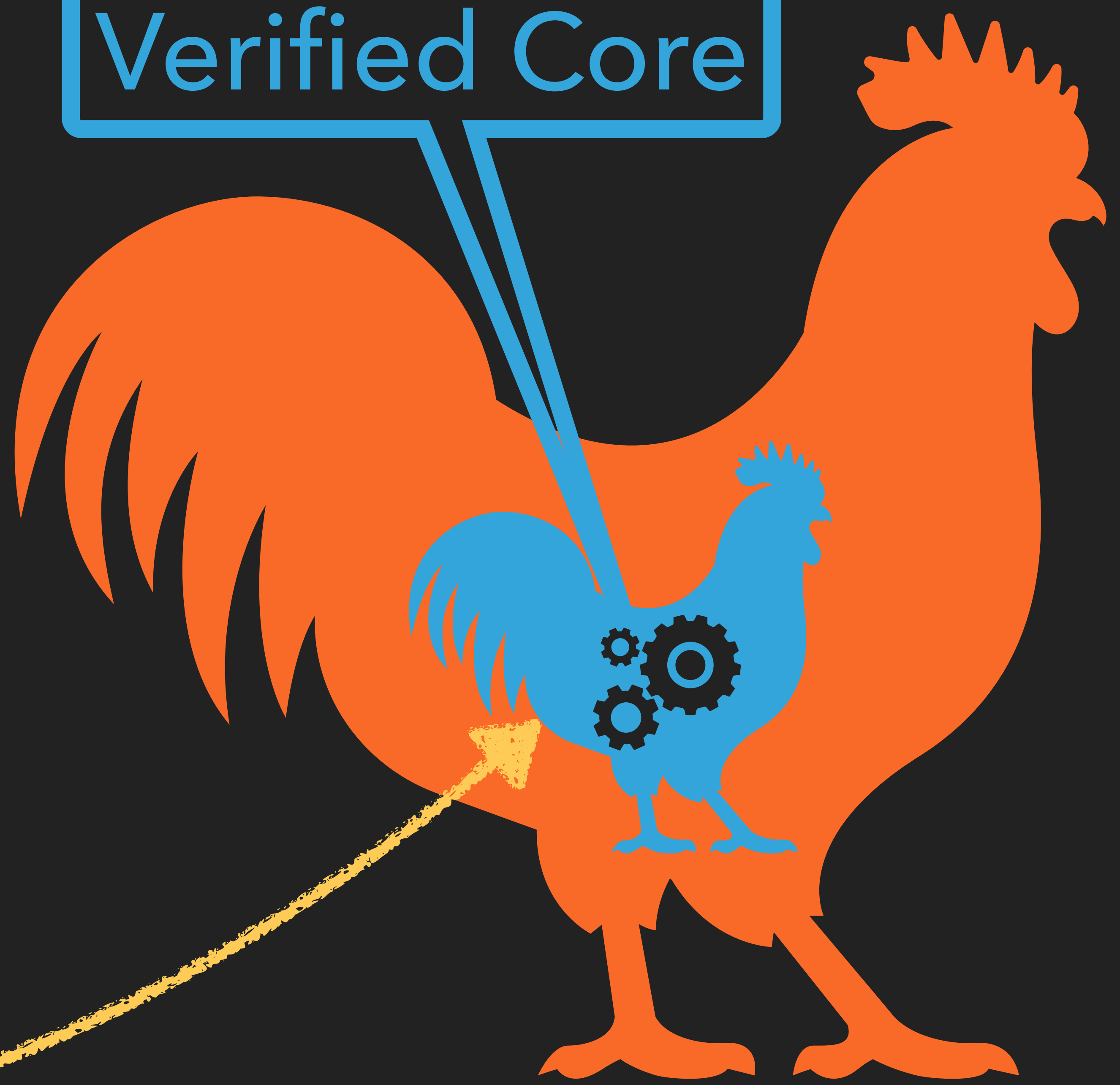
Verified Coq

in

MetaCoq

Verified ℰ

in



Implemented Coq

=

Ideal Coq

Spec: 30kLoC

Proofs: 60kLoC

Comments: 10kLoC

https://metacoq.github.io

Related Work

- Kumar et al., HOL + CakeML (JAR'16)
- Strub et al., Self-Certification of F* starting with Coq (POPL'12)
- Rahli and Anand, NuPRL in Coq (ITP'14)
- Coq en Coq (Barras 1998, 2021)
- Type Theory Should Eat Itself (Chapman)
- Decidability of Conversion