MetaCoq: Towards a Certified Kernel and Extraction for Coq

JFLAs 2024 - Saint-Jacut-de-la-mer, France
January 31st 2024

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

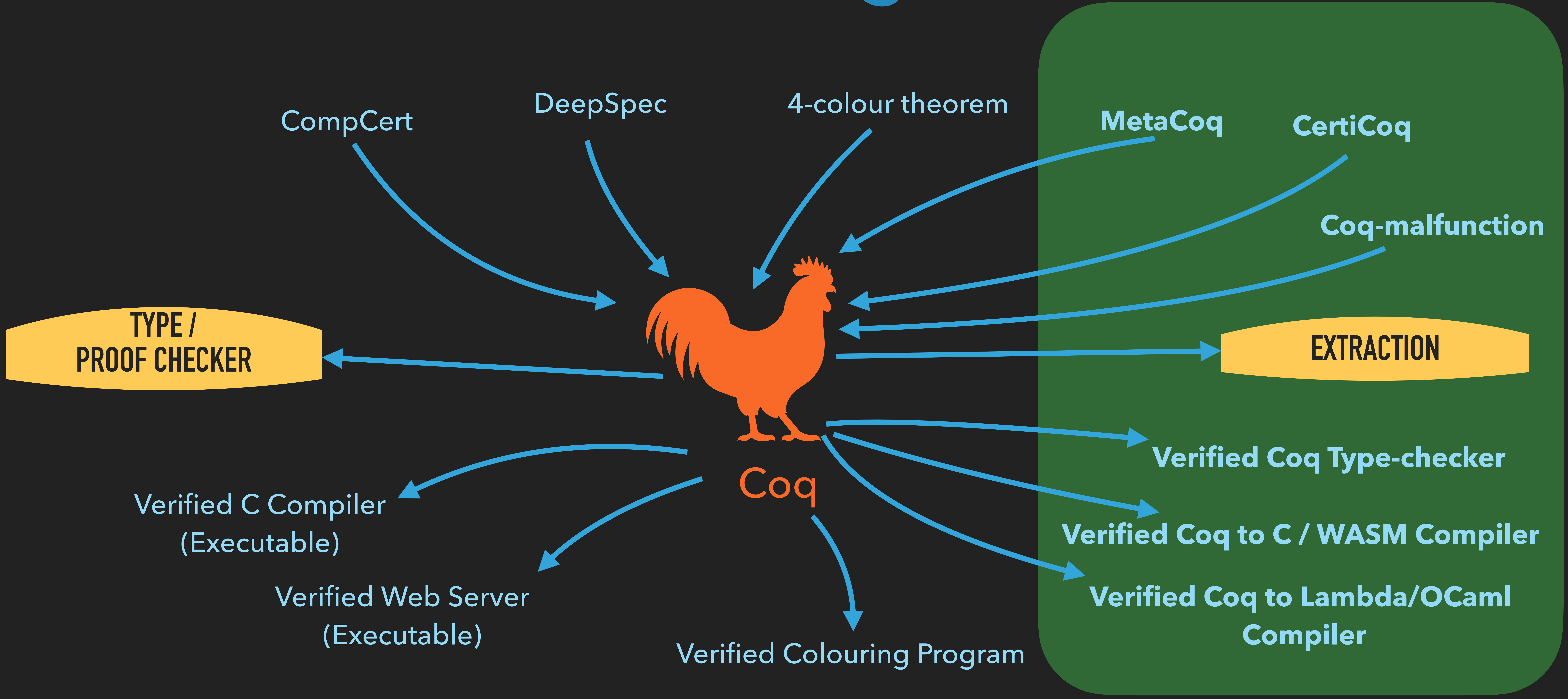
Nicolas **Tabareau**
Inria & LS2N

Théo **Winterhalter**
Inria & LS2N

The MetaCoq 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.

Setting



CompCert

DeepSpec

4-colour theorem

MetaCoq

CertiCoq

Coq-malfunction

**TYPE /
PROOF CHECKER**

EXTRACTION

Coq

Verified C Compiler
(Executable)

Verified Web Server
(Executable)

Verified Colouring Program

Verified Coq Type-checker

Verified Coq to C / WASM Compiler

**Verified Coq to Lambda/OCaml
Compiler**

Summary

I. MetaCoq: meta-theory Coq in Coq

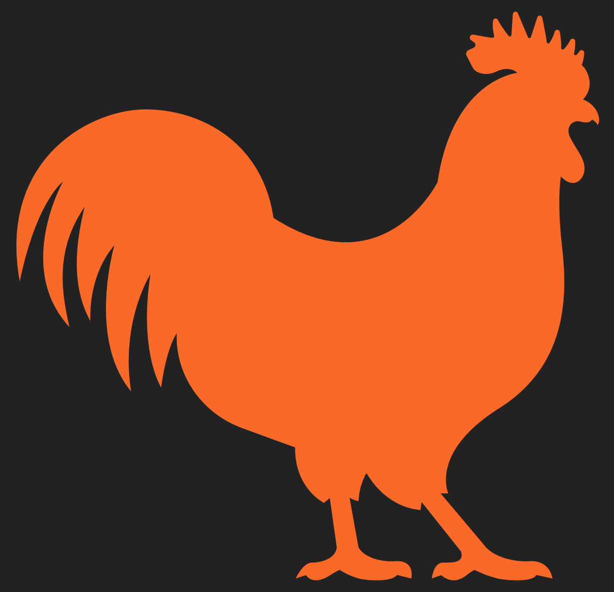
II. Verifying Coq's type-checker

III. Verifying Coq's type-and-proof erasure procedure

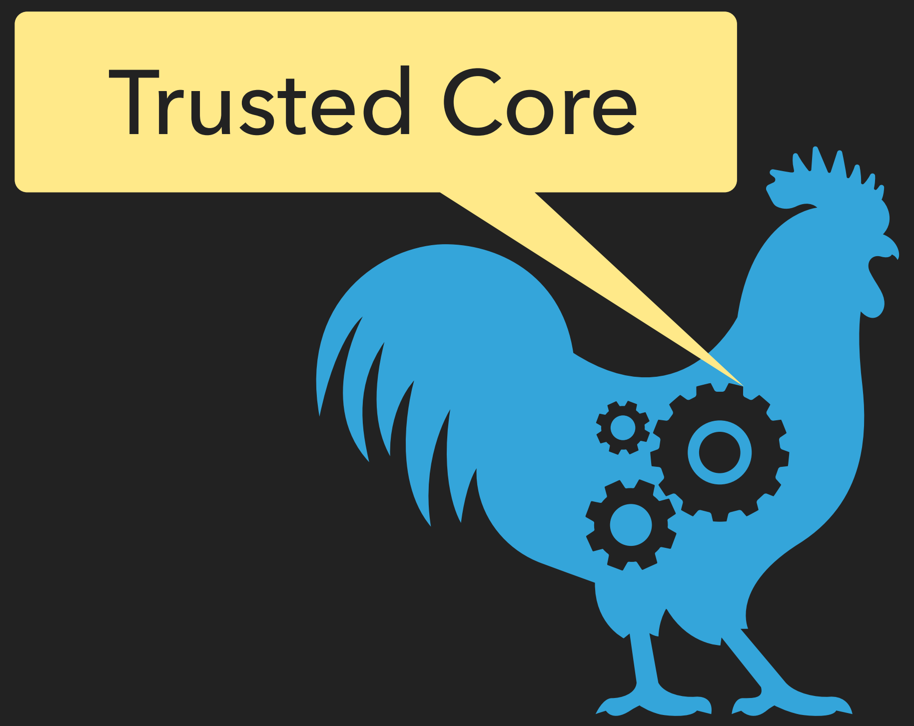
IV. CertiCoq: compilation of extracted programs, from Coq to C & WASM

V. Coq-malfunction: verified extraction for OCaml

What do you trust?



Ideal Coq



Trusted Core

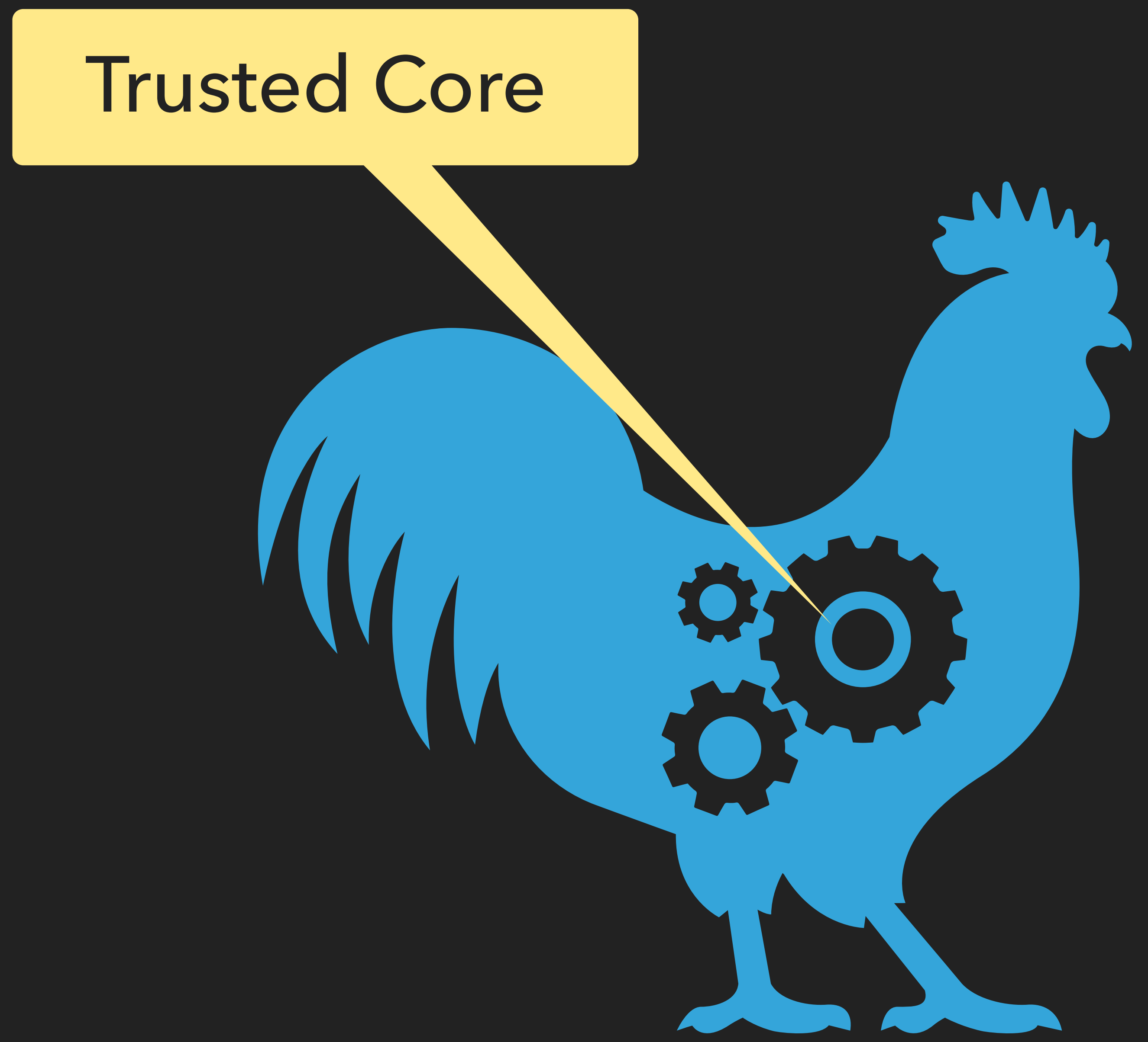
Implemented Coq

What do you trust?

A Dependent Type Checker for PCUIC
(18kLoC, 30+ years)

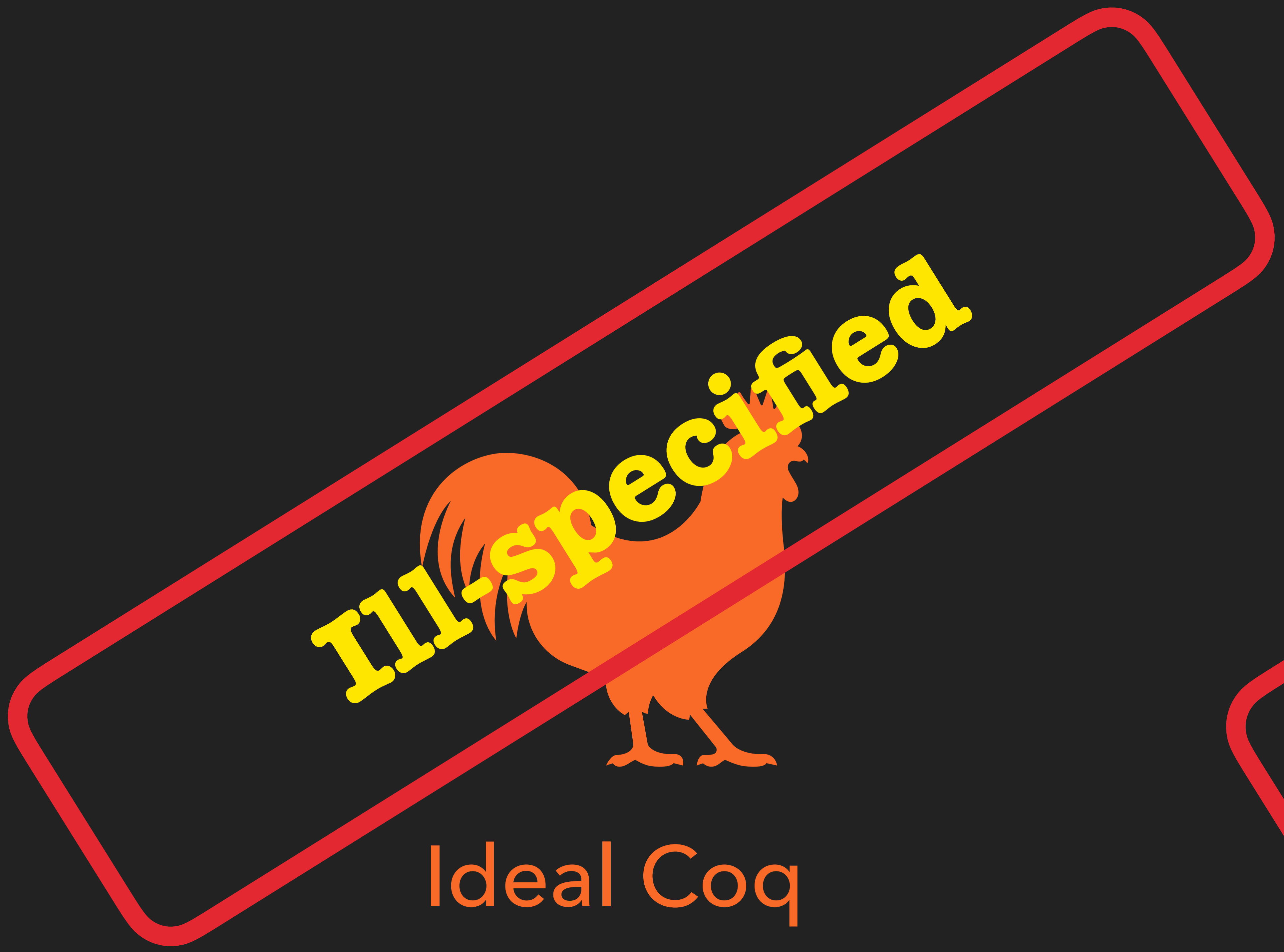
- Inductive Families w/ Guard Checking
- Universe Cumulativity and Polymorphism
- ML-style Module System
- KAM, VM and Native Conversion Checkers

+ OCaml's Compiler and Runtime

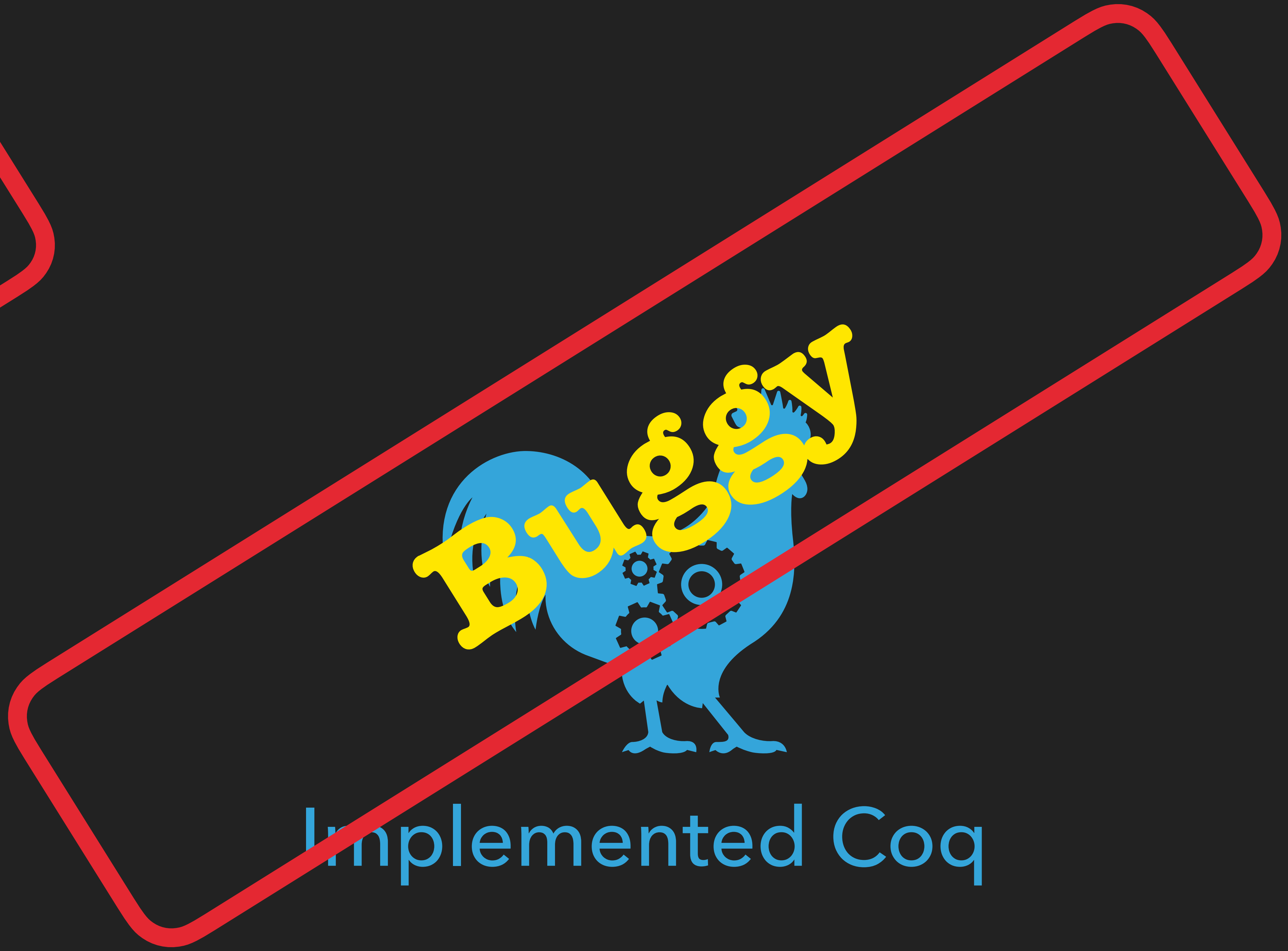


Implemented Coq

The Reality



Ideal Coq



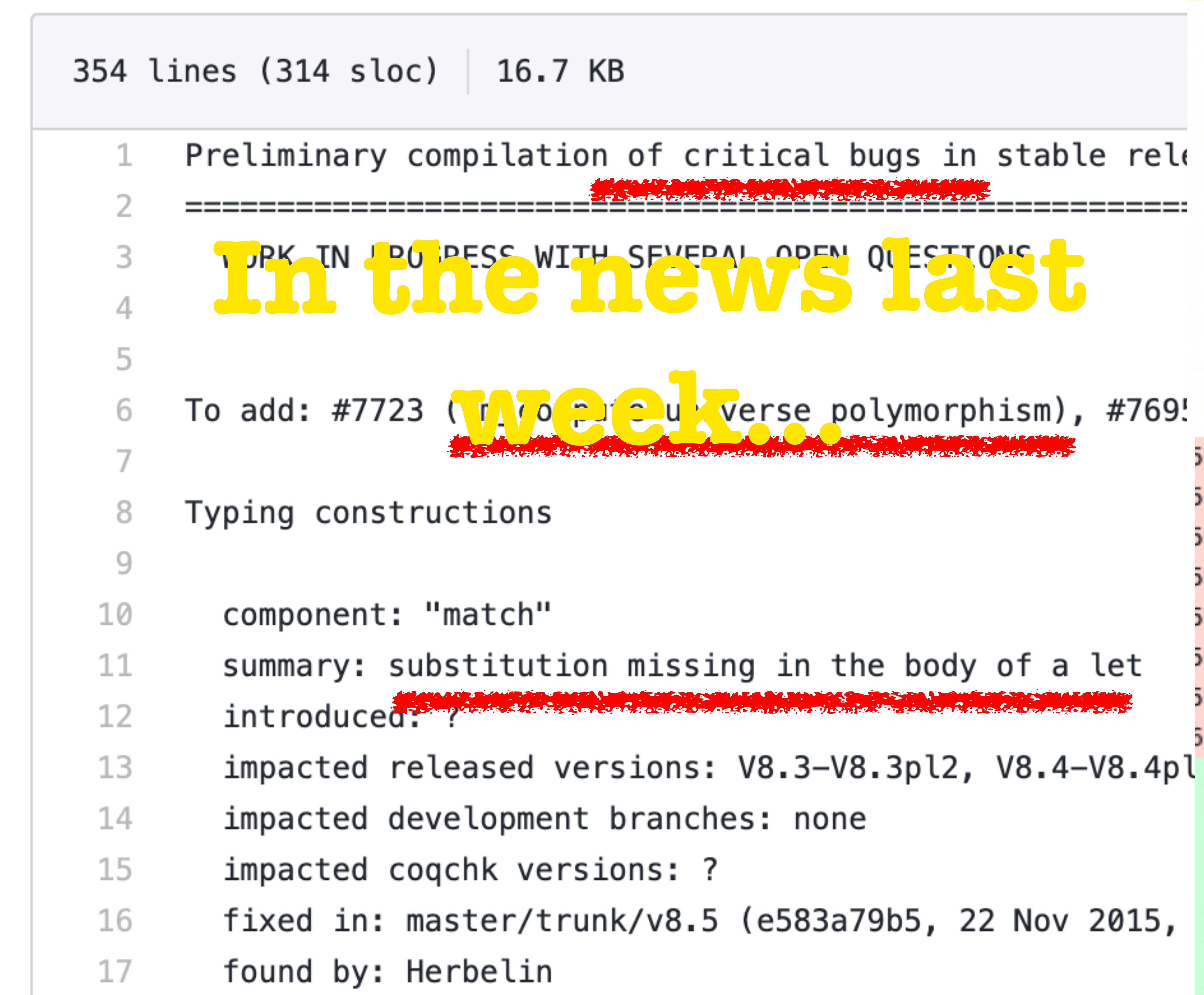
Implemented Coq

Reality Check

Ideal Coq

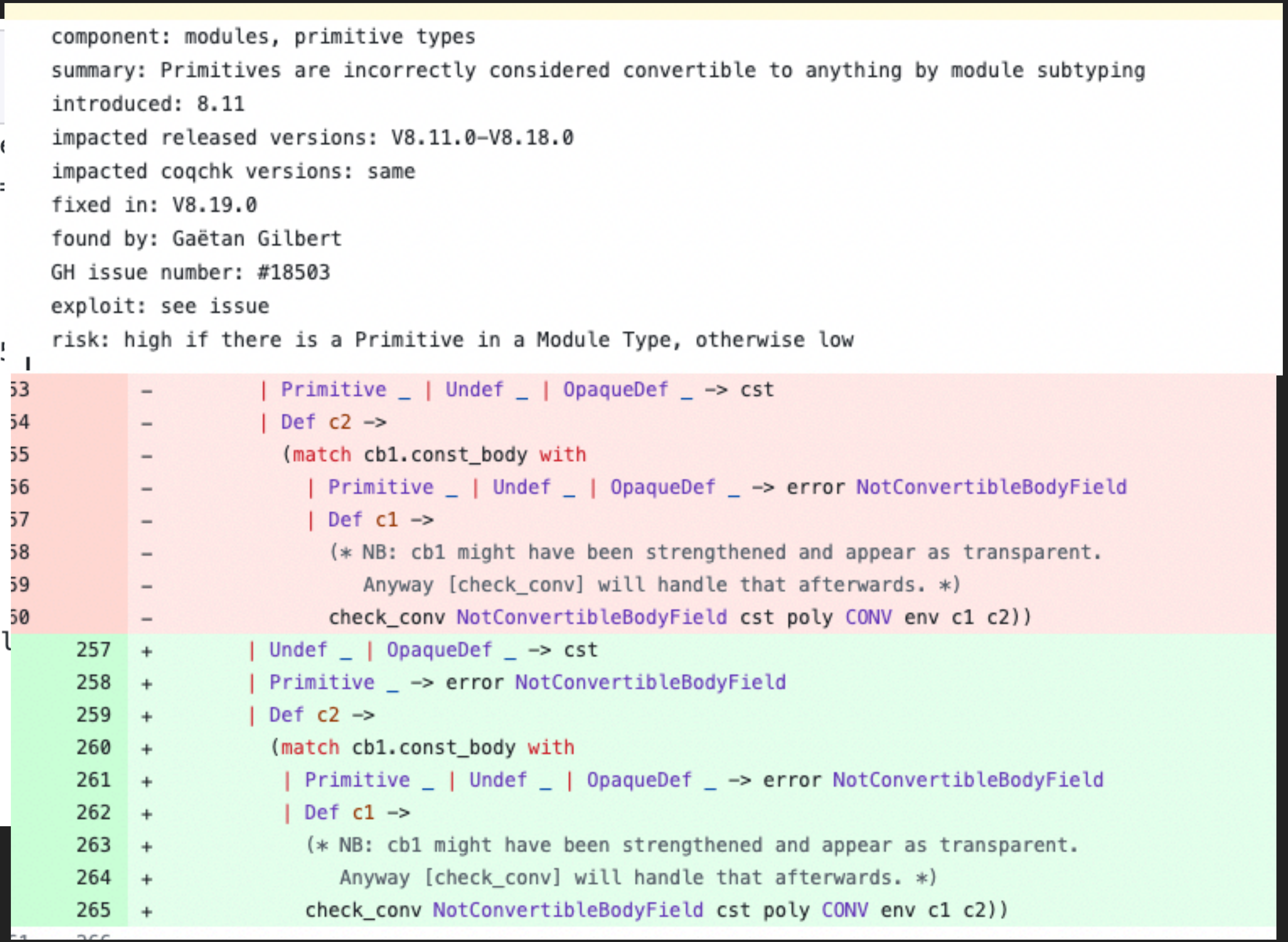
- Reference Manual is semi-formal and partial
- “One feature = n papers/PhDs” where `n : fin 5` e.g. modules, universes, eta-conversion, guard condition, SProp….
- “Discrepancies” with the OCaml implementation
- Combination of features not worked-out in detail. E.g. cumulative inductive types + let-bindings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

Reality Check



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

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


To add: #7723 (template 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
```

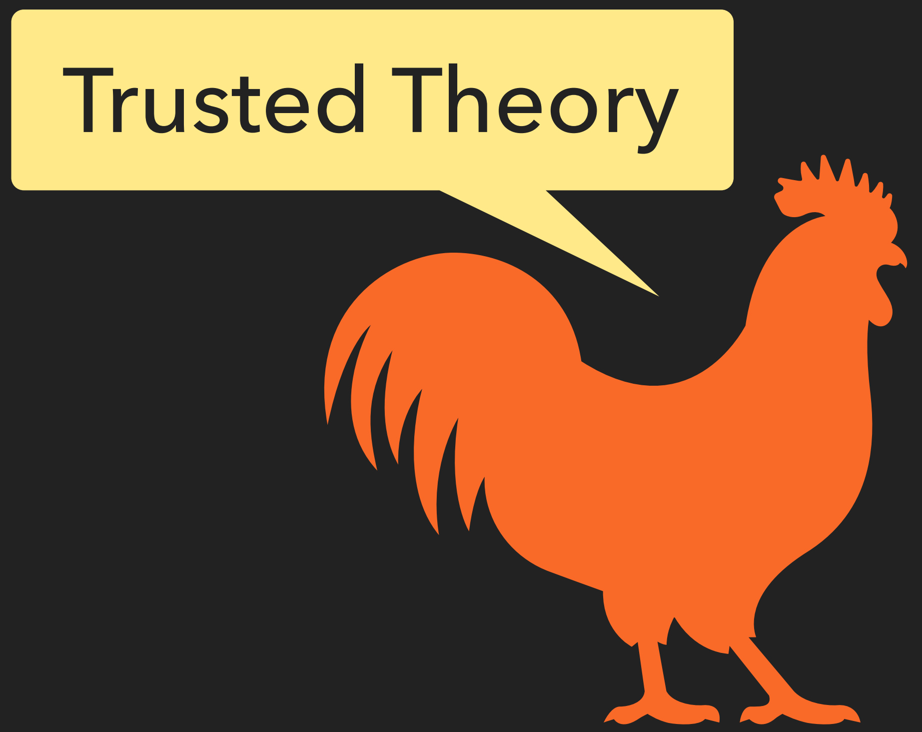
In the news last week…



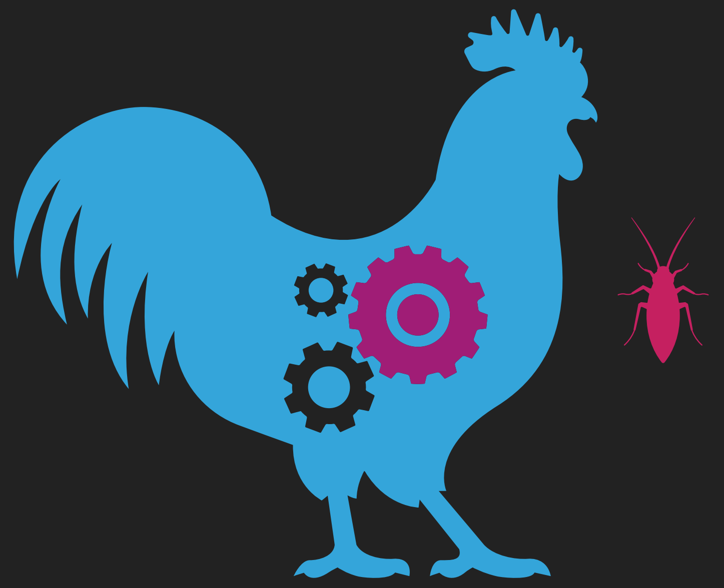
```
component: modules, primitive types
summary: Primitives are incorrectly considered convertible to anything by module subtyping
introduced: 8.11
impacted released versions: V8.11.0-V8.18.0
impacted coqchk versions: same
fixed in: V8.19.0
found by: Gaëtan Gilbert
GH issue number: #18503
exploit: see 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))
```

Our Goal: Improving Trust



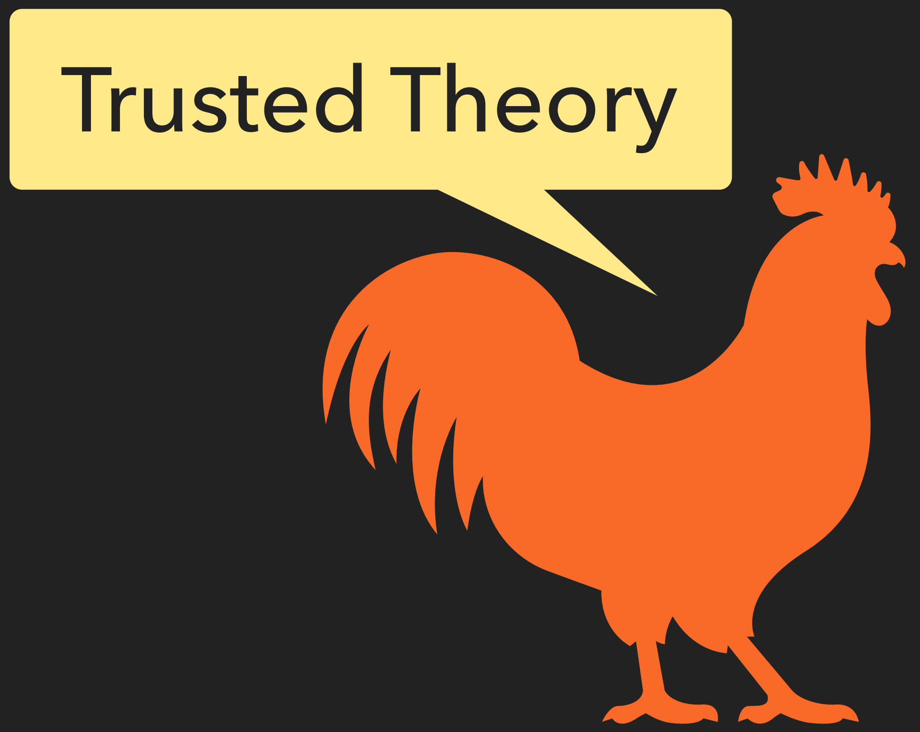
Ideal Coq


~ 1 critical bug every year

Implemented Coq

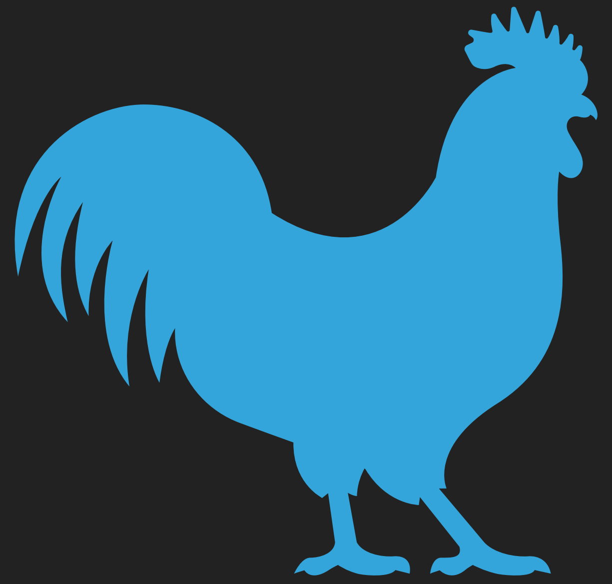
Coq in MetaCoq

Verified metatheory,
sound implementation



Part I: Coq's Calculus PCUIC

↕



Part II: Verified Coq

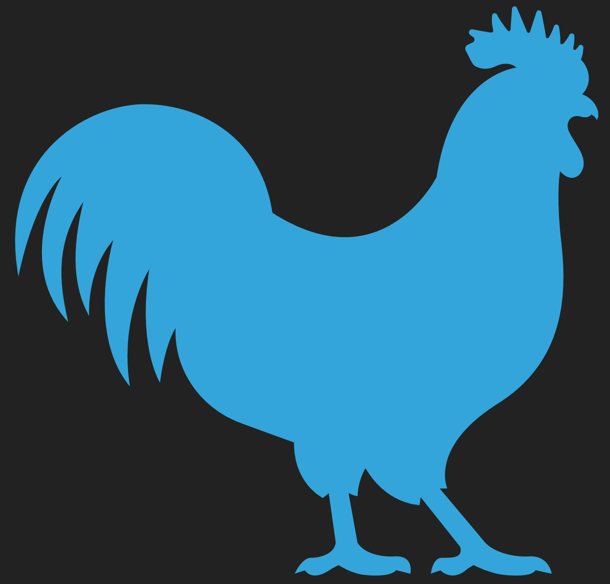
POPL'20

in

MetaCoq

Formalization of
Coq in Coq
ITP'19, JAR'20

in



Implemented Coq

MetaCoq in Practice

A meta-programming library

DEMO!

Part I

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.
```

Specification

Example: Reduction

DEFINITIONS IN CONTEXTS

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

GENERAL SUBSTITUTION

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

STRONG REDUCTION

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

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

$\Sigma ; \Gamma \vdash t \to u, \ t \to^* u$	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 $\iff T \to^* T' \wedge U \to^* U' \wedge T' \leq_\alpha U'$
$\Sigma ; \Gamma \vdash t : T$	Typing
$\mathtt{wf}\ \Sigma, \ \mathtt{wf\_local}\ \Sigma\ \Gamma$	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
- 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 Coq, similar to Agda

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

Specification Global set of consistent constraints, satisfy a valuation in 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

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

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

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

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.

$$\begin{tikzcd}
t \arrow[d, Rrightarrow] \\
\rho(t)
\end{tikzcd}$$

The triangle property

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

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
```

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

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
```

$$\frac{\begin{array}{c}\Sigma \; ; \; \Gamma \vdash t : T \qquad \Sigma \; ; \; \Gamma \vdash u : U \\ \Sigma \vdash u \leq_{\alpha\_noind} t\end{array}}{\Sigma \; ; \; \Gamma \vdash u : T}$$

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).

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

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 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
- Could be inherited by preservation of normalisation from a stronger system with a model

See Martin-Löf à la Coq (CPP’24) for the state of the art!

Part II

Verifying Type-Checking

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

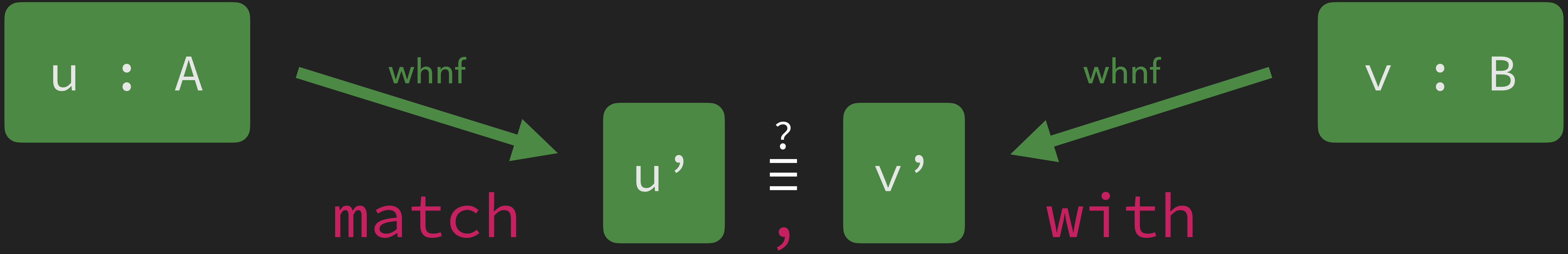
$$u : A \xrightarrow{\text{whnf}} u' \qquad v' \xleftarrow{\text{whnf}} v : B$$

Conversion

Algorithm

Conversion

Algorithm



$$\begin{tikzcd} u : A \arrow[r, "\text{whnf}"] & u' \arrow[r, phantom, "\overset{?}{\equiv}"] & v' & v : B \arrow[l, "\text{whnf}"'] \end{tikzcd}$$

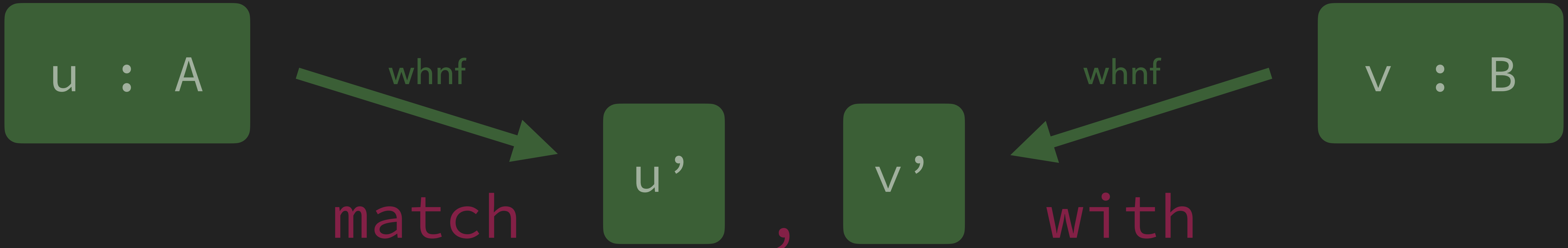
`match` u', v' `with`

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

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

Conversion

Completeness



$u : A \xrightarrow{\text{whnf}} u'$

$v : B \xrightarrow{\text{whnf}} v'$

match u' , v' with

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

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

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

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

we conclude

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

using inversion lemmata and confluence

Conversion

$$\begin{tikzcd} u : A \arrow[r, "\mathrm{whnf}"] & u' & v' & v : B \arrow[l, "\mathrm{whnf}"'] \end{tikzcd}$$

`match` u' , v' `with`

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

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

■ ■ ■

Weak head reduction

Objective

Input	Output
`u`	`v`
`term`	`term`

Weak head reduction

Objective

Input

`u`

`term`

Output

`v`

`term`

`u -> v`

`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 ⟶ 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`

$$\texttt{foo 0} \longrightarrow \texttt{(λ(x:nat).x) 0} \longrightarrow \texttt{0}$$

Weak head reduction

Termination

Input

u

Output

v

u -> v

Weak head reduction

Termination

Input

$u \quad \pi_1$

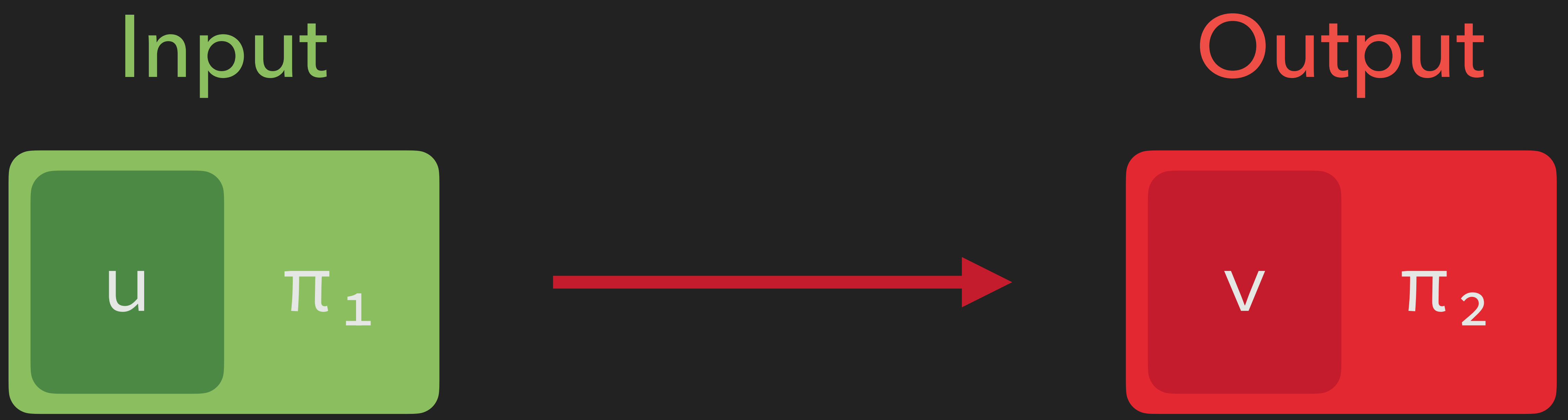
Output

$v \quad \pi_2$

u -> v

Weak head reduction

Termination



Input

Output

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`

`(λ(x:nat).x) 0 ⟶ 0`

Weak head reduction

Termination

`foo 0 ⟶ (λ(x:nat).x) 0`

`foo 0 ⊐ foo`

`(λ(x:nat).x) 0 ⟶ 0`

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

`foo 0` ⟶ `(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` ⟶ `0`

Lexicographic order of -> and ⊏

Weak head reduction

Termination

```
p.1
```

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

`p.1`

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination

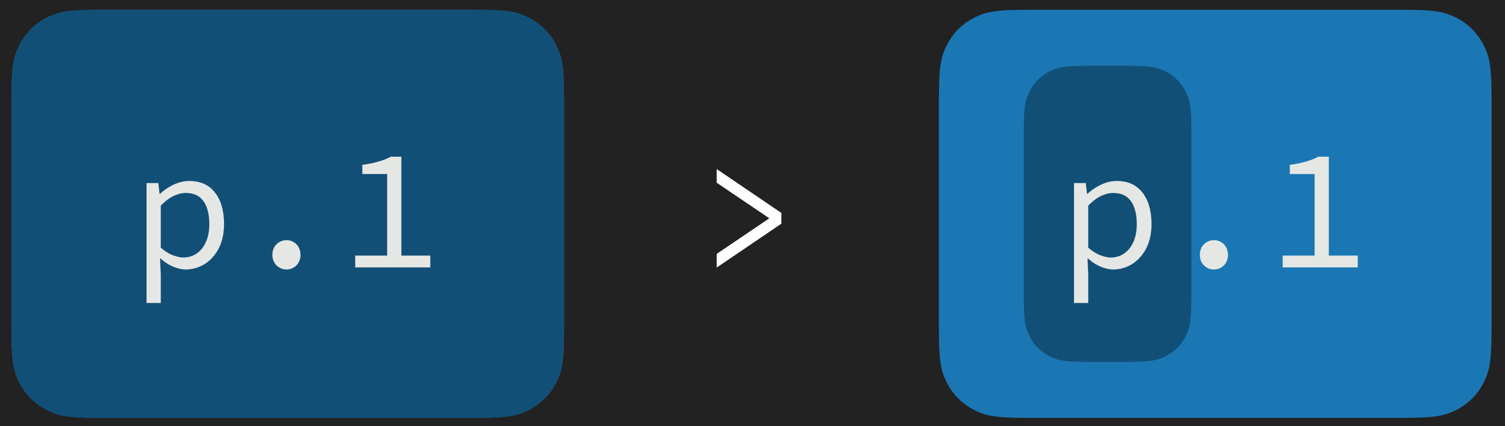
`p.1`   `p.1`

but `p.1 ≠ p`

1 Lexicographic order of `->` and ⊏

Weak head reduction

Termination



`p.1 > p.1`

`and p.1 = p.1`

1 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 ⊏

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
```

▼

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

~~Lexicographic order of → and ⊏~~

Weak head reduction

Termination

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

Weak head reduction

Termination

2

Lexicographic order of -> and an order on positions

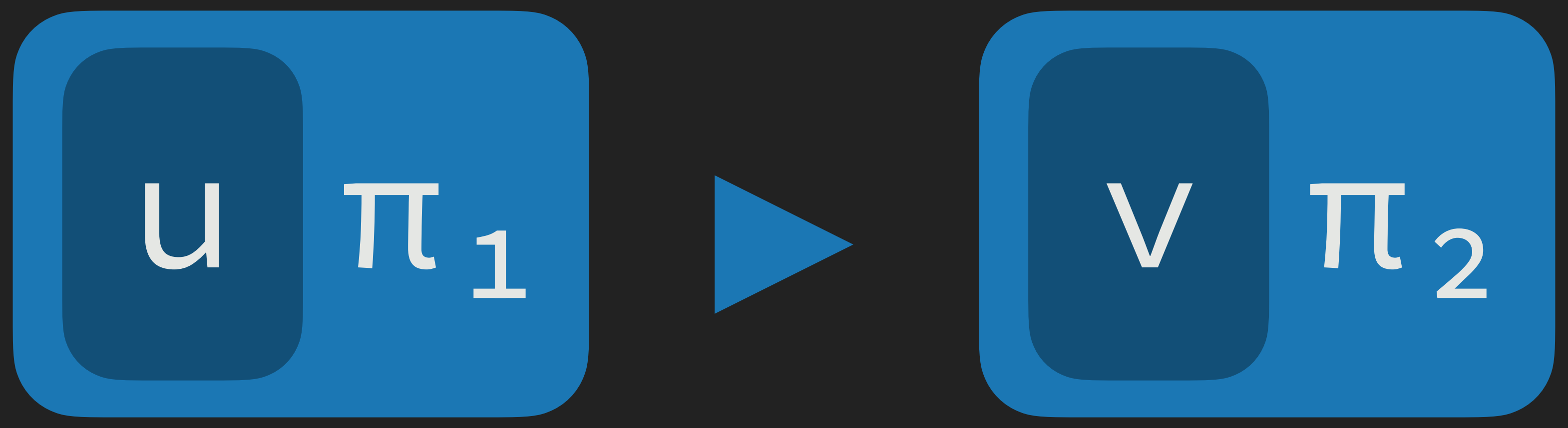
Weak head reduction

Termination

2 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

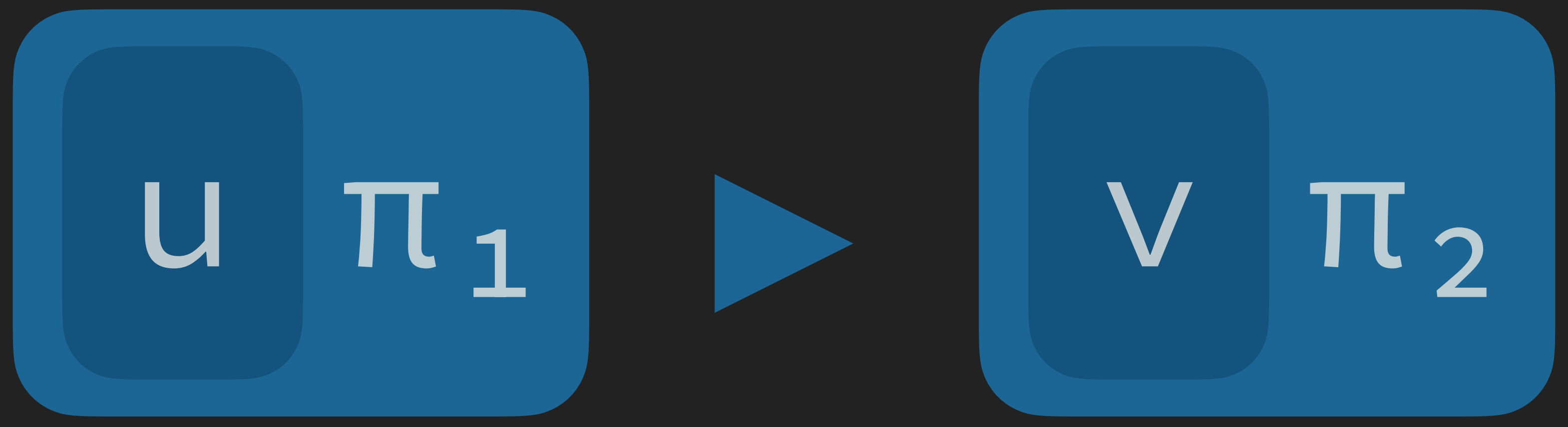
$$\boxed{\boxed{u}\ \pi_1} \blacktriangleright \boxed{\boxed{v}\ \pi_2}$$

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

Lexicographic order of `->` and an order on positions

Weak head reduction

Termination

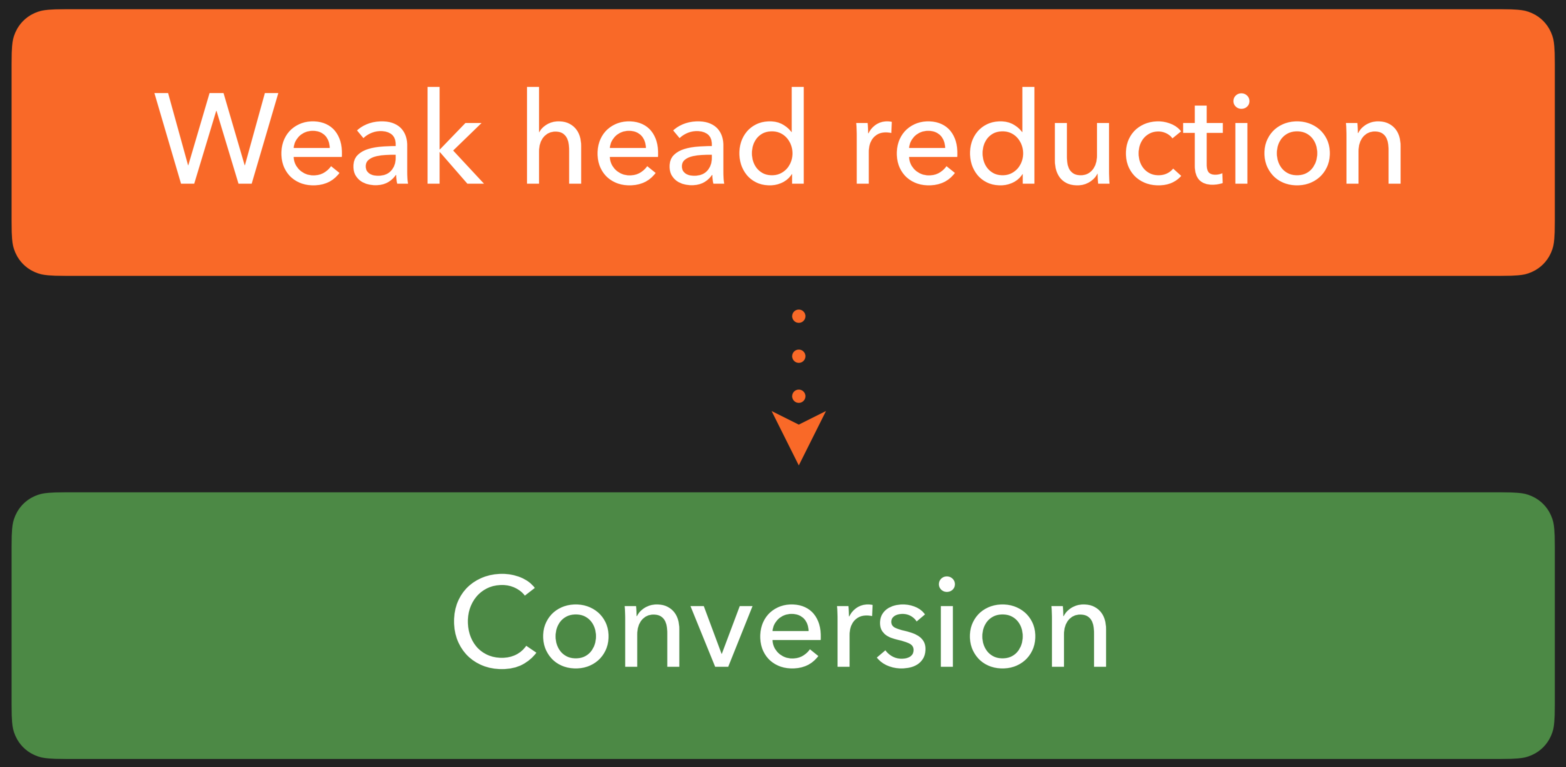


$$u\ \pi_1 \triangleright v\ \pi_2$$

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

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

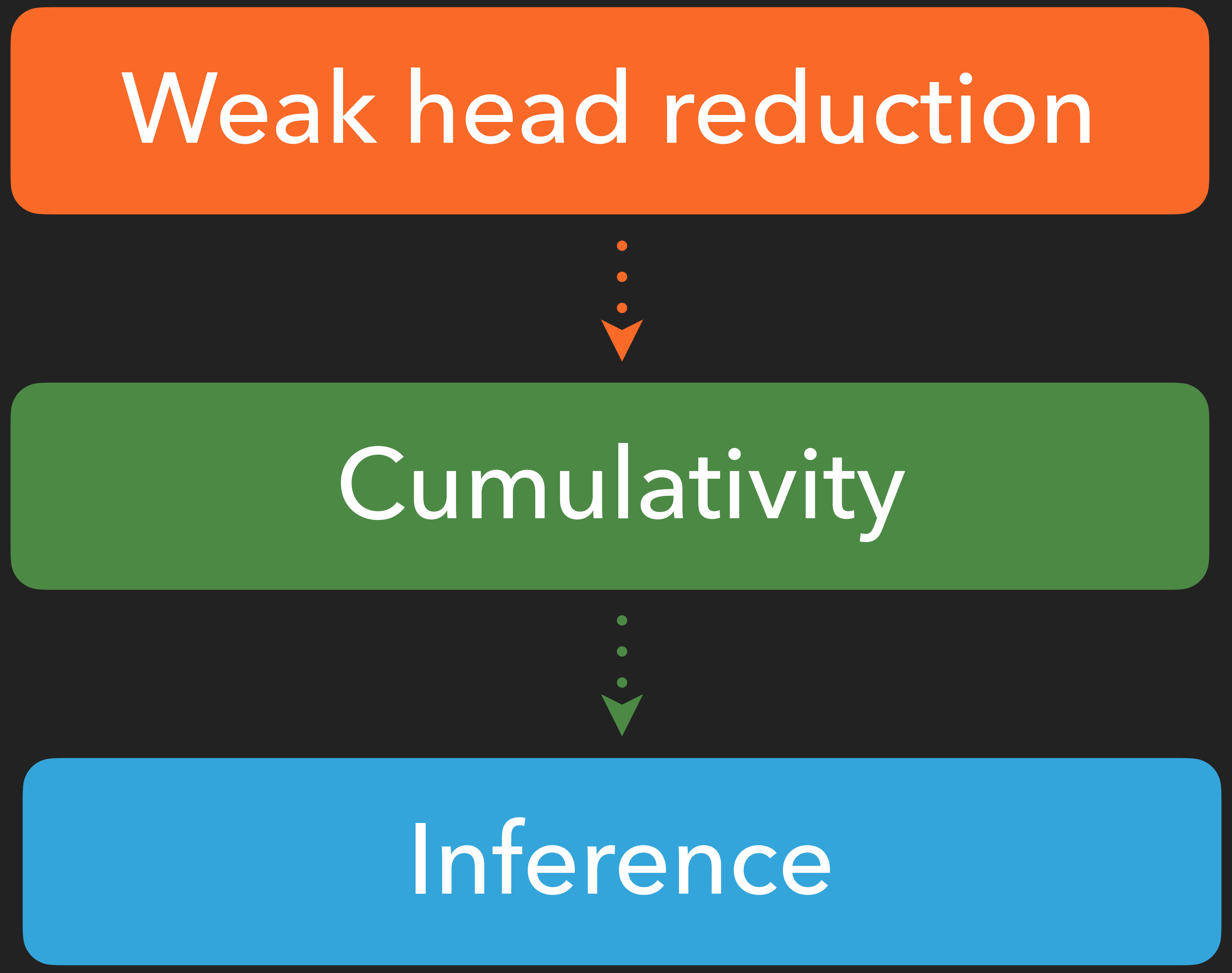
Type Checking



Weak head reduction

Conversion

Type Checking

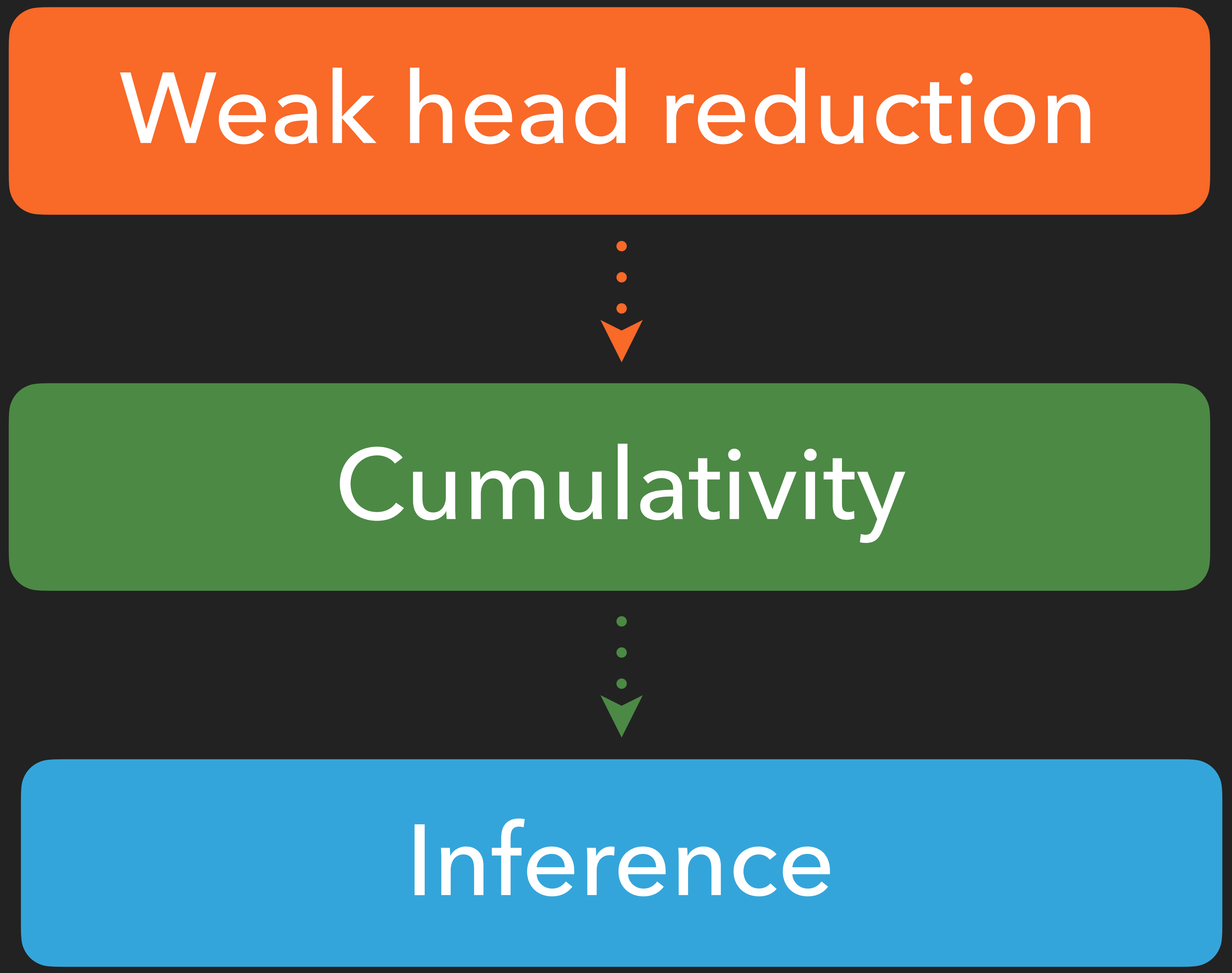


Weak head reduction

Cumulativity

Inference

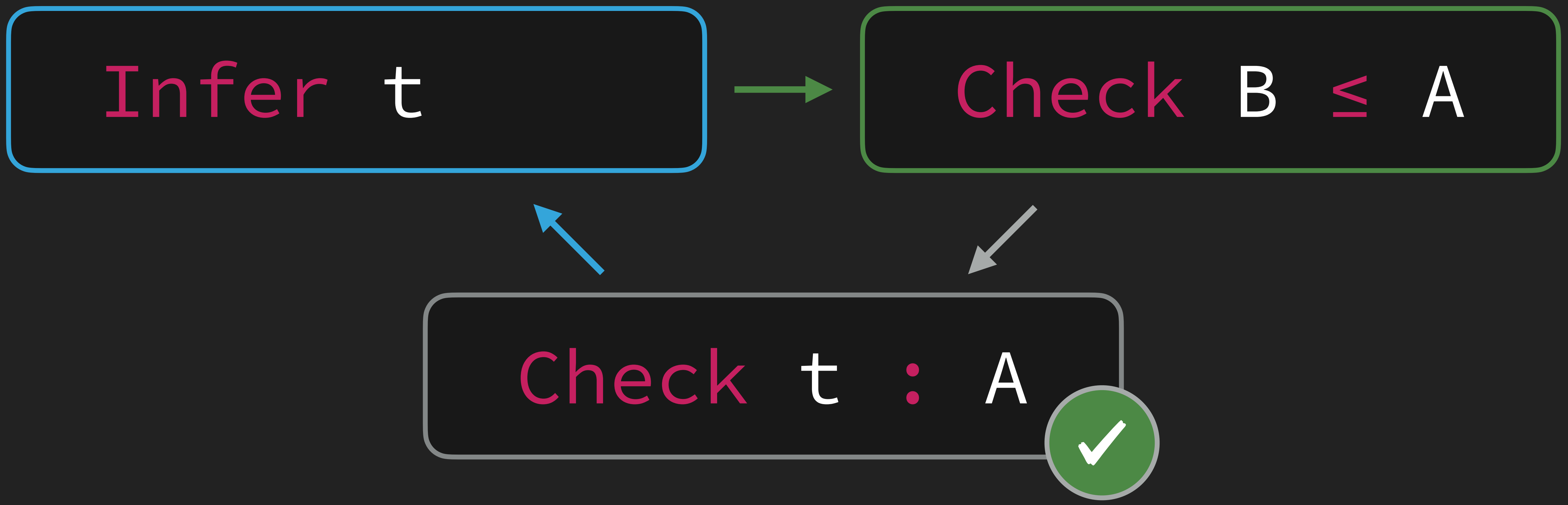
Type Checking



Weak head reduction

Cumulativity

Inference

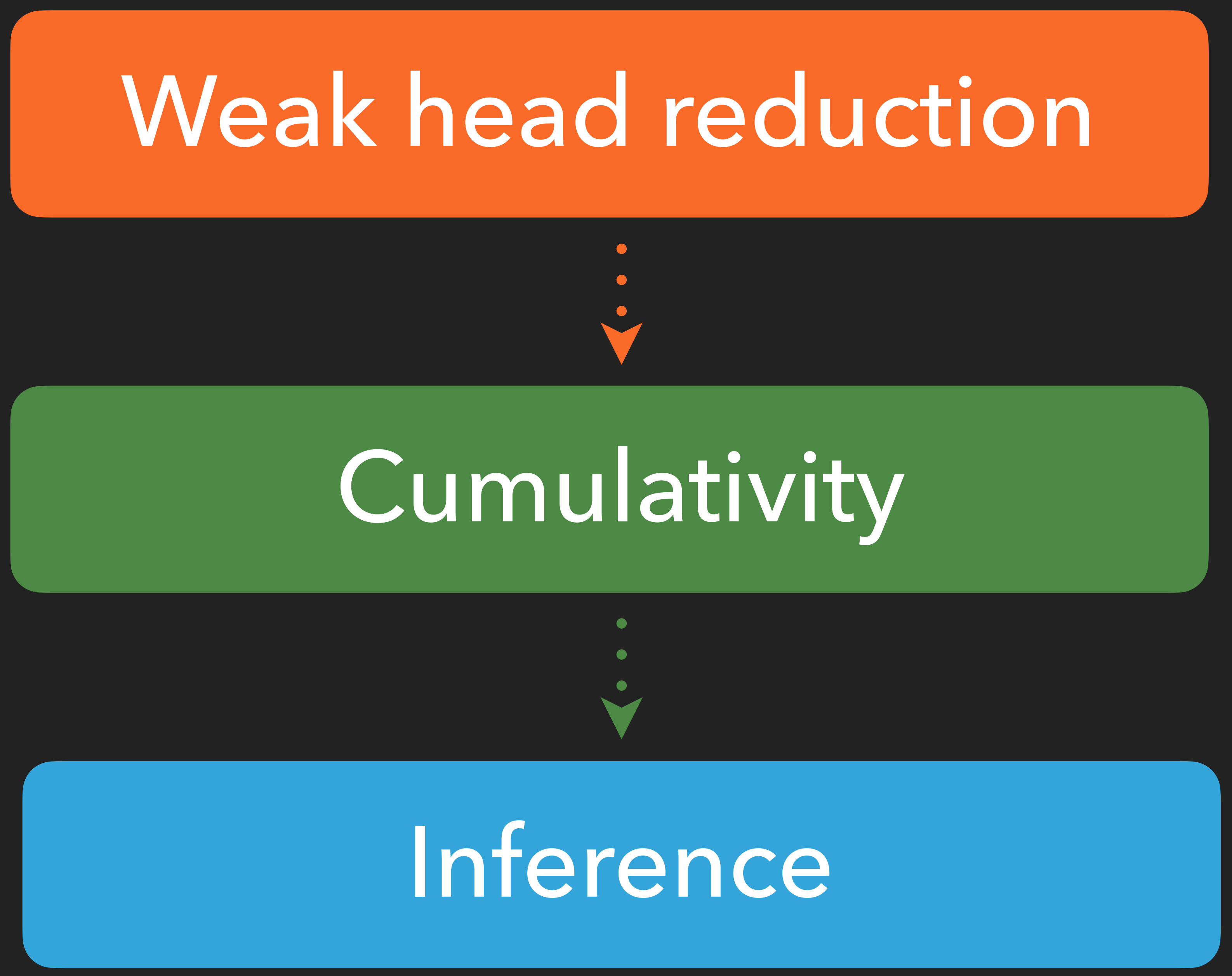


`Infer t`

`Check B ≤ A`

`Check t : A` ✓

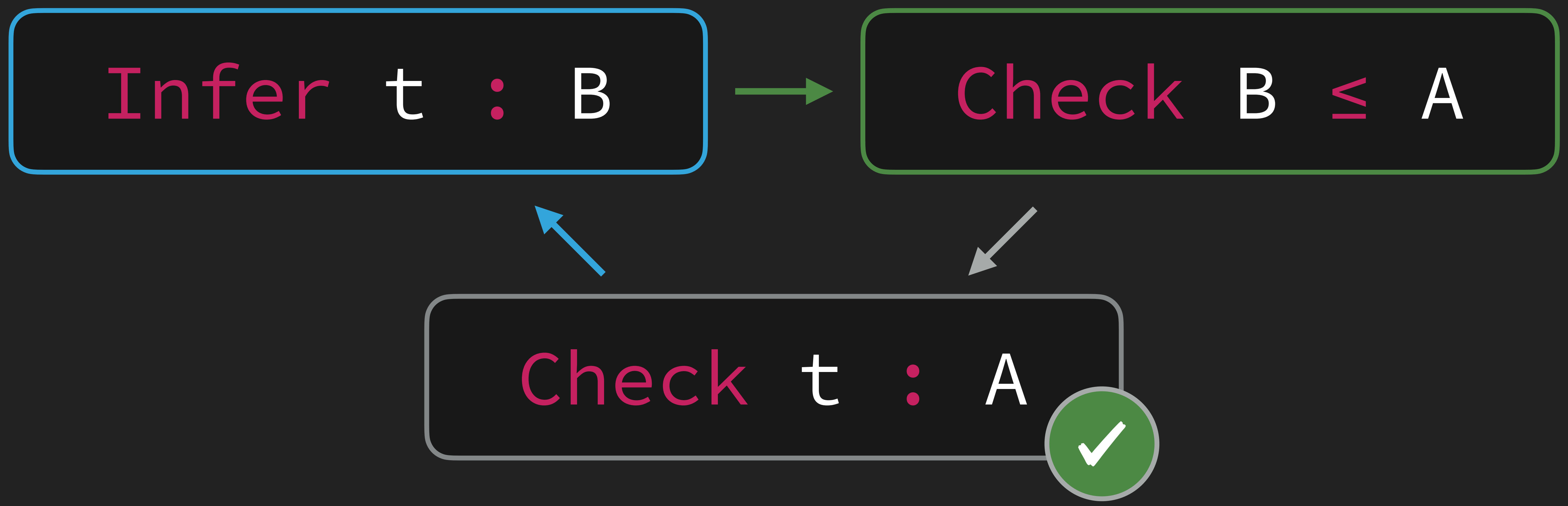
Type Checking



Weak head reduction

Cumulativity

Inference



Infer t : B

Check B ≤ A

Check t : A ✓

```
MetaCoq 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

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 Derivations

$\Sigma \, ; \, \Gamma \vdash t \triangleright T$ (Σ, Γ and t are inputs, T output)

- Inference: T is the minimal type of t (and is well-formed)
- Checking has a single rule here (the only rule that is not directed by the syntax of the term t)

$$\frac{\Sigma \, ; \, \Gamma \vdash t \triangleright T \qquad \Sigma \, ; \, \Gamma \vdash T \le U}{\Sigma \, ; \, \Gamma \vdash t \triangleleft U}\ \text{Cumul}$$

Typing algorithm

```
infer : forall Σ Γ t,
  { T : term | Σ ; Γ ⊢ t > T} +
  ~ { T : term | Σ ; Γ ⊢ t > T}

check : forall Σ Γ t T,
 { Σ ; Γ ⊢ t < T } + { ~ Σ ; Γ ⊢ t < T }
```

+ proofs of equivalence:

```
infer_check : Σ ; Γ ⊢ t > T -> Σ ; Γ ⊢ t < T

check_typing : Σ ; Γ ⊢ t < T -> Σ ; Γ ⊢ t : T

typing_check : Σ ; Γ ⊢ t : T -> Σ ; Γ ⊢ t < T
```

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

Part III

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:

  `Ɛ (t : Type) = ☐`

- Proof erasure:

  `Ɛ (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.
```

`Ɛ (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

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

$\vdash t : nat$

=> $\vdash t \rightarrow n$ /\ n irreducible (strong normalization)

=> $\vdash t \rightarrow n : nat$ /\ $n \in \mathbb{N}$ (subject reduction and canonicity)

=> $\vdash t \rightarrow_{cbv} n$ /\ $n \in \mathbb{N}$ (standardisation)

=> $\mathcal{E}(t) \rightarrow_{cbv} \mathcal{E}(n) = n$ (erasure correctness + extracted naturals are equivalent to naturals)

Erasure Correctness

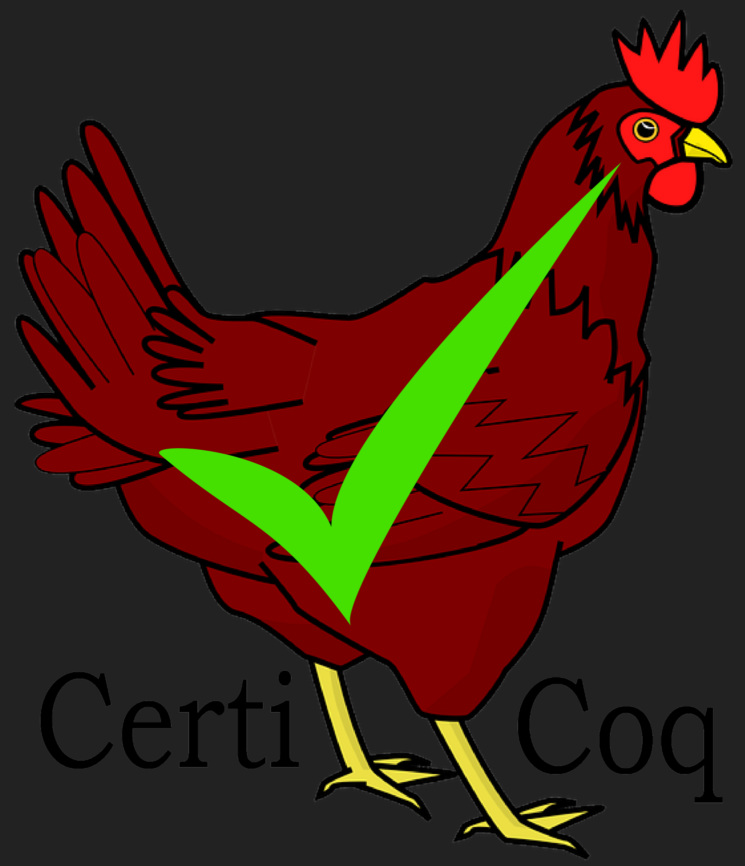
First define a non-deterministic erasure relation, then define:

```
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

Part IV
CertiCoq



Compiler Correctness

Forward Simulation Proofs

$$\begin{tikzcd}
t \arrow[r, "\mathrm{cbv}"] \arrow[d, "\mathrm{CertiCoq}"'] & v \arrow[d, dotted, leftrightarrow, "\text{Observational Equivalence}"] & (\text{in } \Lambda^{\square,\mathrm{match},\mathrm{fix},\mathrm{cofix}}) \\
t' \arrow[r, "\mathrm{cbv}"] & \exists v' & (\text{in } C)
\end{tikzcd}$$

With Canonicity and SN:

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

CertiCoq

- Strip parameters (e.g. nil instead of nil nat)
- Let-bind definitions in the global environment
- Compile case-analysis to switch + projections
- ANF or CPS translation
- Closure conversion
- Defunctionalization (first-order program)
- Inlining and shrinking (remove administrative redexes)
- Generation of C code, linked with a certified garbage collector

We get back a C program with the same results as the Coq program (but optimised behavior)

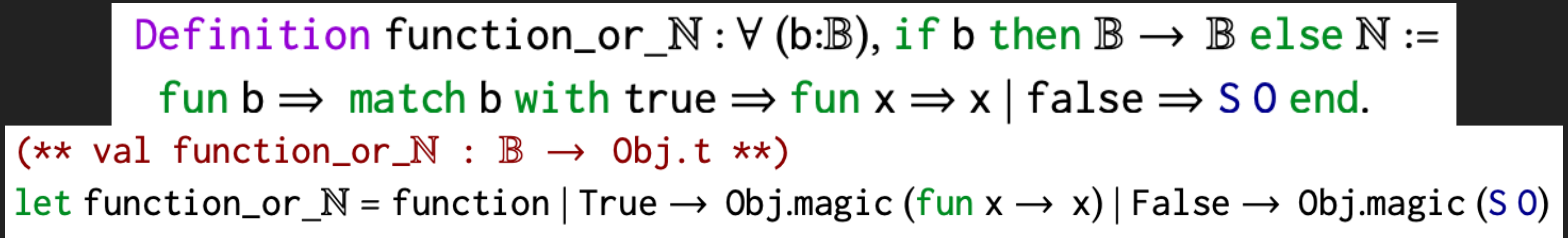
$$\begin{tikzcd}
t \arrow[r, phantom, "\rightarrow_{cbv}"] \arrow[d] & v \arrow[d, dotted, leftrightarrow, "\text{Observational Equivalence}"] \\
t' \arrow[r, phantom, "\rightarrow_{cbv}"] & \exists v'
\end{tikzcd}$$

CertiCoq

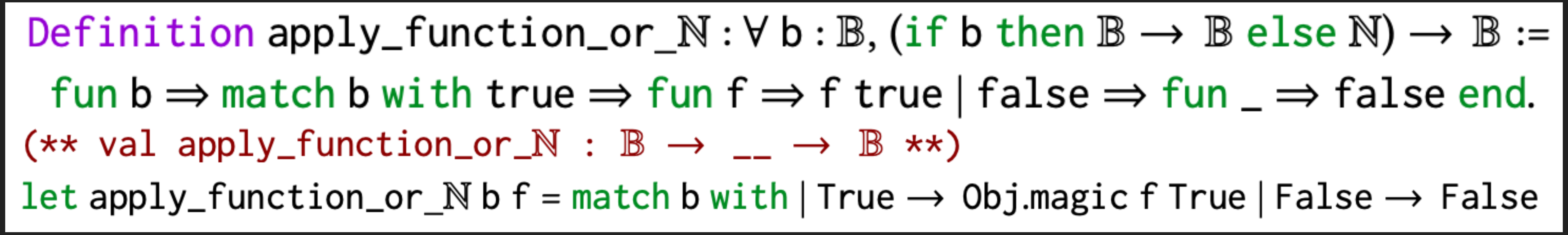
- Supports “Extract Constant” to realize Coq axioms in C (e.g. primitive integers and floating point values)
- VeriFFI project to link verified C code with CertiCoq-compiled Coq programs (e.g. efficient imperative data structures)
- From C-light, we can use the certified CompCert compiler to produce certified assembly code, or LLVM/gcc (standard C compilers)
- Alternative target: WASM

Part V
coq-malfunction

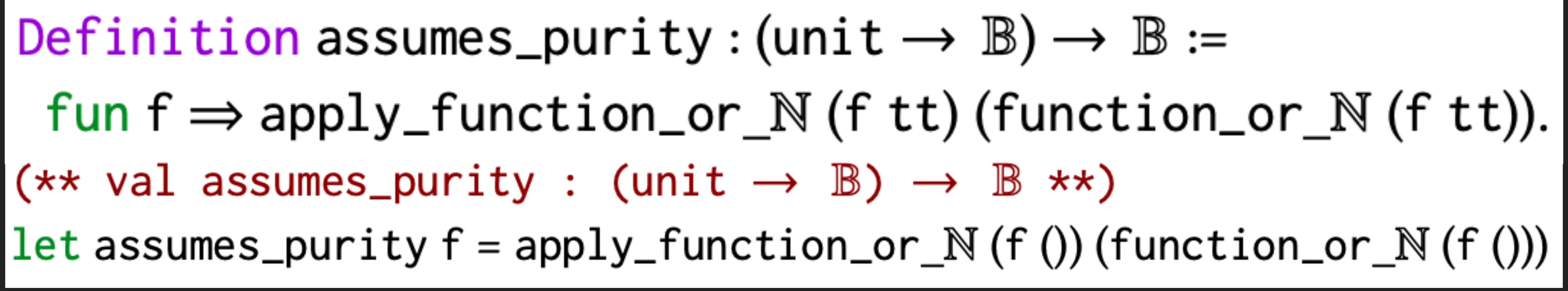
Coq's current extraction



```
Definition function_or_ℕ : ∀ (b:𝔹), if b then 𝔹 → 𝔹 else ℕ :=
  fun b ⇒  match b with true ⇒ fun x ⇒ x | false ⇒ S O end.
(** val function_or_ℕ : 𝔹 → Obj.t **)
let function_or_ℕ = function | True → Obj.magic (fun x → x) | False → Obj.magic (S O)
```

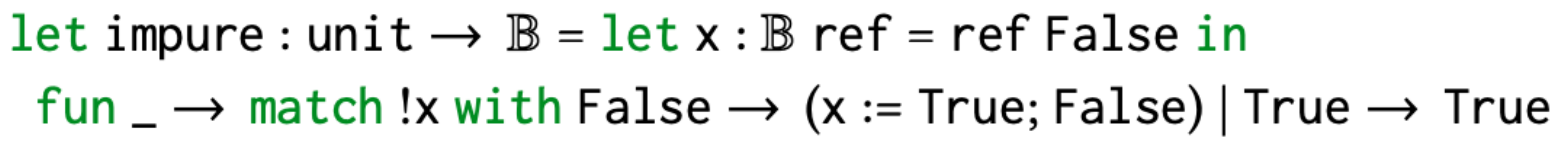


```
Definition apply_function_or_ℕ : ∀ b : 𝔹, (if b then 𝔹 → 𝔹 else ℕ) → 𝔹 :=
  fun b ⇒ match b with true ⇒ fun f ⇒ f true | false ⇒ fun _ ⇒ false end.
(** val apply_function_or_ℕ : 𝔹 → __ → 𝔹 **)
let apply_function_or_ℕ b f = match b with | True → Obj.magic f True | False → False
```

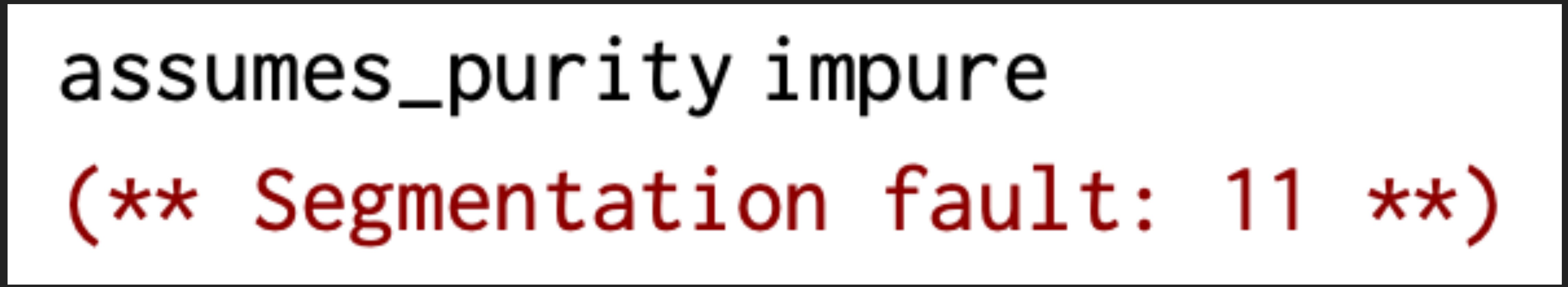


```
Definition assumes_purity : (unit → 𝔹) → 𝔹 :=
  fun f ⇒ apply_function_or_ℕ (f tt) (function_or_ℕ (f tt)).
(** val assumes_purity : (unit → 𝔹) → 𝔹 **)
let assumes_purity f = apply_function_or_ℕ (f ()) (function_or_ℕ (f ()))
```

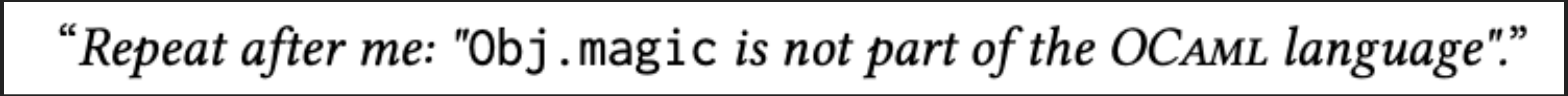
Coq’s current extraction



```
let impure : unit → 𝔹 = let x : 𝔹 ref = ref False in
 fun _ → match !x with False → (x := True; False) | True → True
```



```
assumes_purity impure
(** Segmentation fault: 11 **)
```



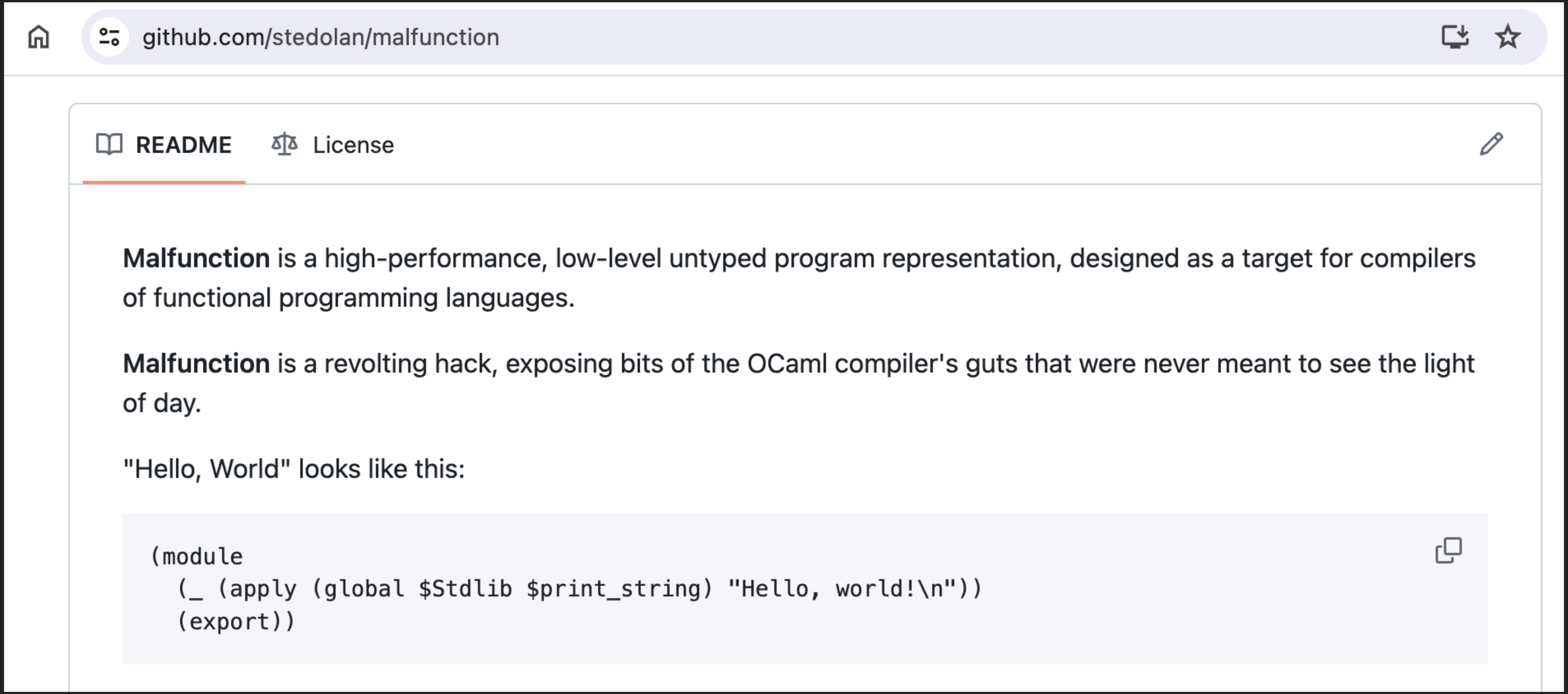
“*Repeat after me:* "`Obj.magic` *is not part of the OCaml language*".”

Xavier Leroy

The way out: away with types!

- Typed extraction to a weaker type system is bound to be unsafe
- Restrict correctness to a subset of types that can be faithfully extracted
- Only first-order inductive types without indices (e.g. nat) and functions between them (no higher-order) can appear in the extracted **interface**.
- Extracted **implementations** can do anything, in an untyped way
- Provide a strong **interoperability** theorem: any OCaml use of the extracted Coq value will be safe

Malfunction



Malfunction is a high-performance, low-level untyped program representation, designed as a target for compilers of functional programming languages.

Malfunction is a revolting hack, exposing bits of the OCaml compiler's guts that were never meant to see the light of day.

"Hello, World" looks like this:

```
(module
  (_ (apply (global $Stdlib $print_string) "Hello, world!\n"))
  (export))
```

Malfunction & coq-malfunction

- AST of untyped OCaml terms (including refs, …)
  Using HOAS, tricky mutual fix point representation
- Compiler from malfunction to cmxs (ocaml object files), providing a trusted .mli interface.
- A reference interpreter ported to Coq (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

$$\begin{tikzcd} t \arrow[r, "\rightarrow_{cbv}"] \arrow[d, "\texttt{coq-malfunction}"'] & v \quad (\text{in } \Lambda^{\square}\texttt{,match,fix,cofix}) \arrow[d, leftrightarrow, dotted, "\text{Observational Equivalence}"] \\ t' \arrow[r, "\rightarrow_{cbv}"] & \exists v' \quad (\text{in } \texttt{OCaml/malfunction}) \end{tikzcd}$$

With Canonicity and SN:

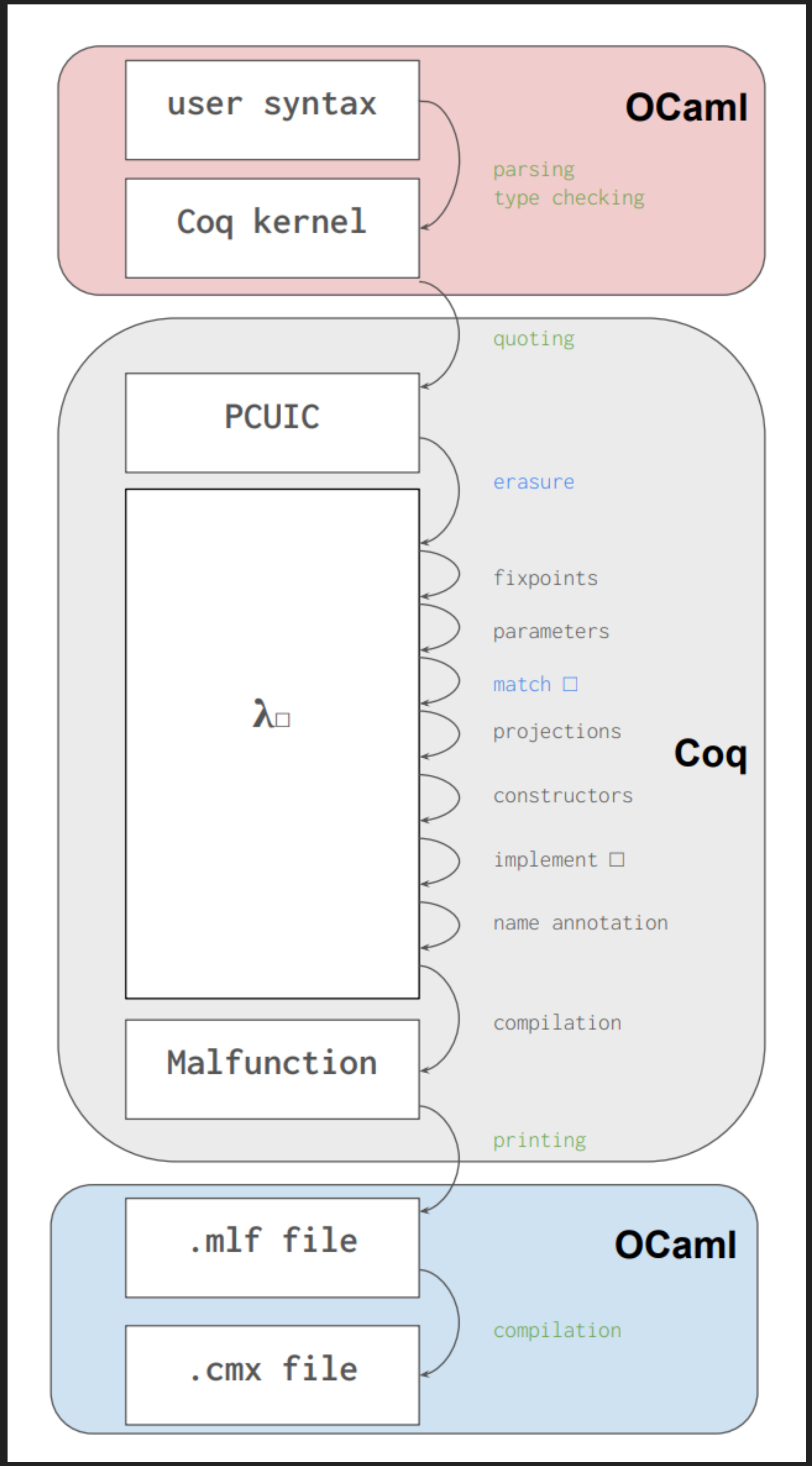
$$\begin{aligned} & \vdash t : \mathtt{nat} \\ \Rightarrow\ & \vdash t \rightarrow n : \mathtt{nat} \qquad (n \in \mathbb{N}) \\ \Rightarrow\ & t \rightarrow_{cbv} n : \mathtt{nat} \\ \Rightarrow\ & \texttt{coq-malfunction}\ (t) \rightarrow_{cbv} n \end{aligned}$$

Separate compilation

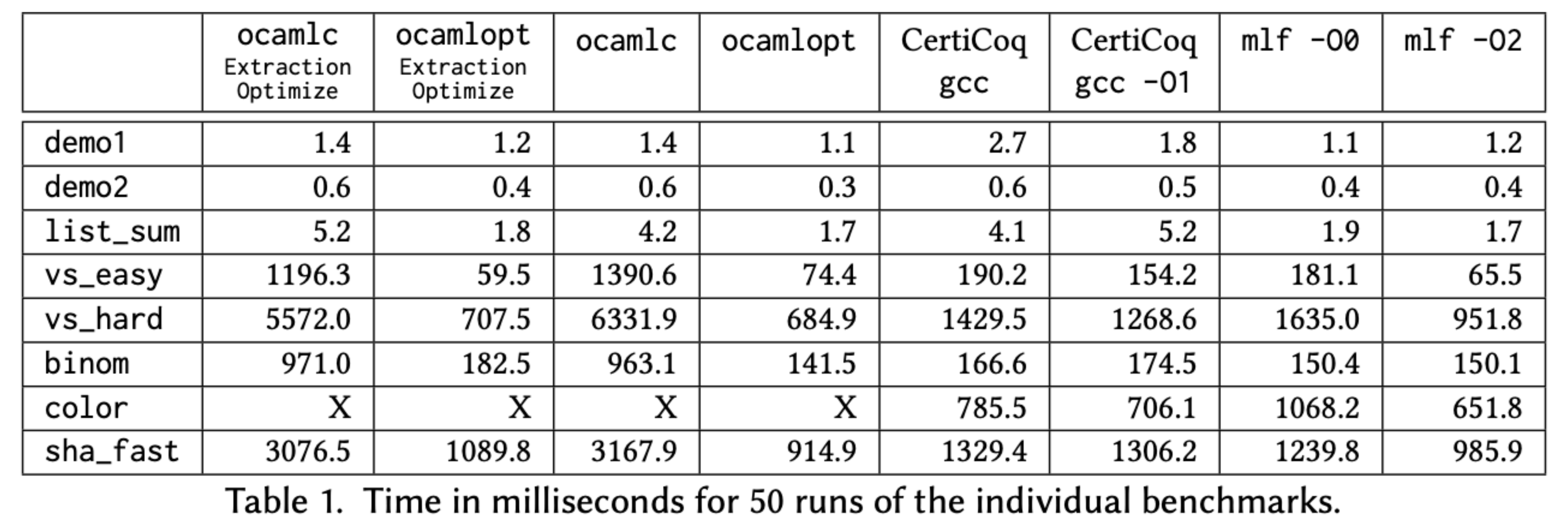
$$\frac{\vdash \texttt{t : nat -> nat} \qquad \vdash \texttt{u : nat} \qquad \texttt{t u} \rightarrow_{\texttt{cbv}} \texttt{n}}{\texttt{Mapply (coq-malfunction t) (coq-malfunction u)} \rightarrow_{\texttt{cbv}} \texttt{n}}$$

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

coq-malfunction pipeline



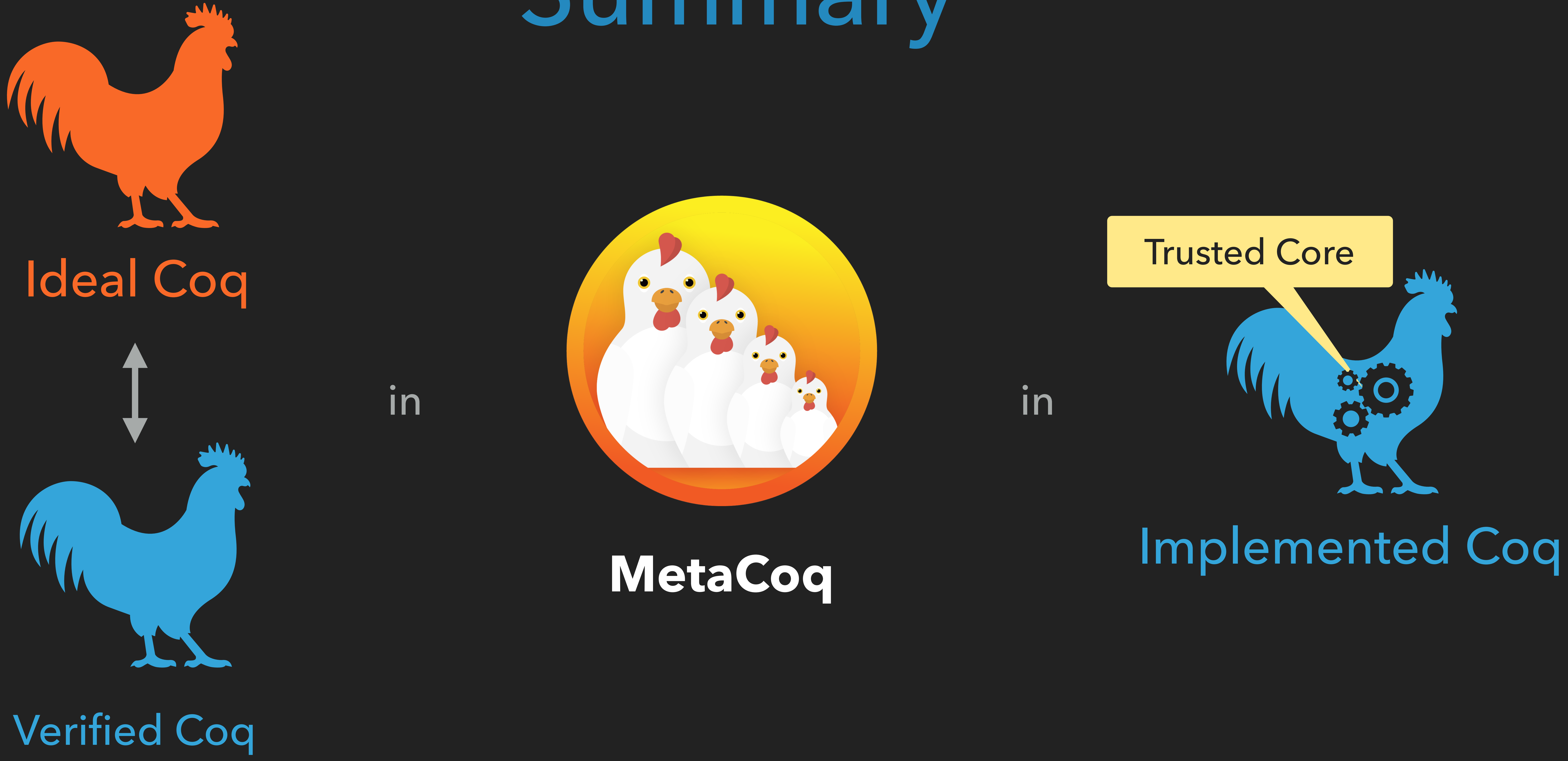
Benchmarks



	ocamlc Extraction Optimize	ocamlopt Extraction Optimize	ocamlc	ocamlopt	CertiCoq gcc	CertiCoq gcc -O1	mlf -O0	mlf -O2
demo1	1.4	1.2	1.4	1.1	2.7	1.8	1.1	1.2
demo2	0.6	0.4	0.6	0.3	0.6	0.5	0.4	0.4
list_sum	5.2	1.8	4.2	1.7	4.1	5.2	1.9	1.7
vs_easy	1196.3	59.5	1390.6	74.4	190.2	154.2	181.1	65.5
vs_hard	5572.0	707.5	6331.9	684.9	1429.5	1268.6	1635.0	951.8
binom	971.0	182.5	963.1	141.5	166.6	174.5	150.4	150.1
color	X	X	X	X	785.5	706.1	1068.2	651.8
sha_fast	3076.5	1089.8	3167.9	914.9	1329.4	1306.2	1239.8	985.9

Table 1. Time in milliseconds for 50 runs of the individual benchmarks.

Summary



Ideal Coq

Verified Coq

in

MetaCoq

in

Trusted Core

Implemented Coq

Summary

Verified Core

Verified Coq

```
MetaCoq Check infer.
```

in

MetaCoq

in

Implemented Coq

=

Ideal Coq

Spec:	80kLoC
Proofs:	120kLoC
Comments:	30kLoC

Verified $\mathcal{E}$ + CertiCoq

```
CertiCoq Compile infer.
```

Going further

MetaCoq

- MetaCoq also includes translations (WIP parametricity translation proof, derivation of principles for inductives)
- WIP integration of SProp, rewrite rules (also in Coq!)
- See metacoq.github.io for documentation, papers and examples
- Part of the Coq platform

Takeaways

MetaCoq

Verified Core

Implemented Coq

=

Ideal Coq

- MetaCoq formalizes the metatheory and proof-checking algorithm Coq in Coq
- Verified extraction and CertiCoq allow to produce verified C code from any Coq program. Safe interoperability with OCaml is possible.
- Verified erasure + CertiCoq + CompCert allow to extract from MetaCoq an efficient, certified proof-checker

MetaCoq

Specified & Verified