Cambium seminar

Coq Coq Correct!

Verification of Type Checking and Erasure for Coq, in Coq

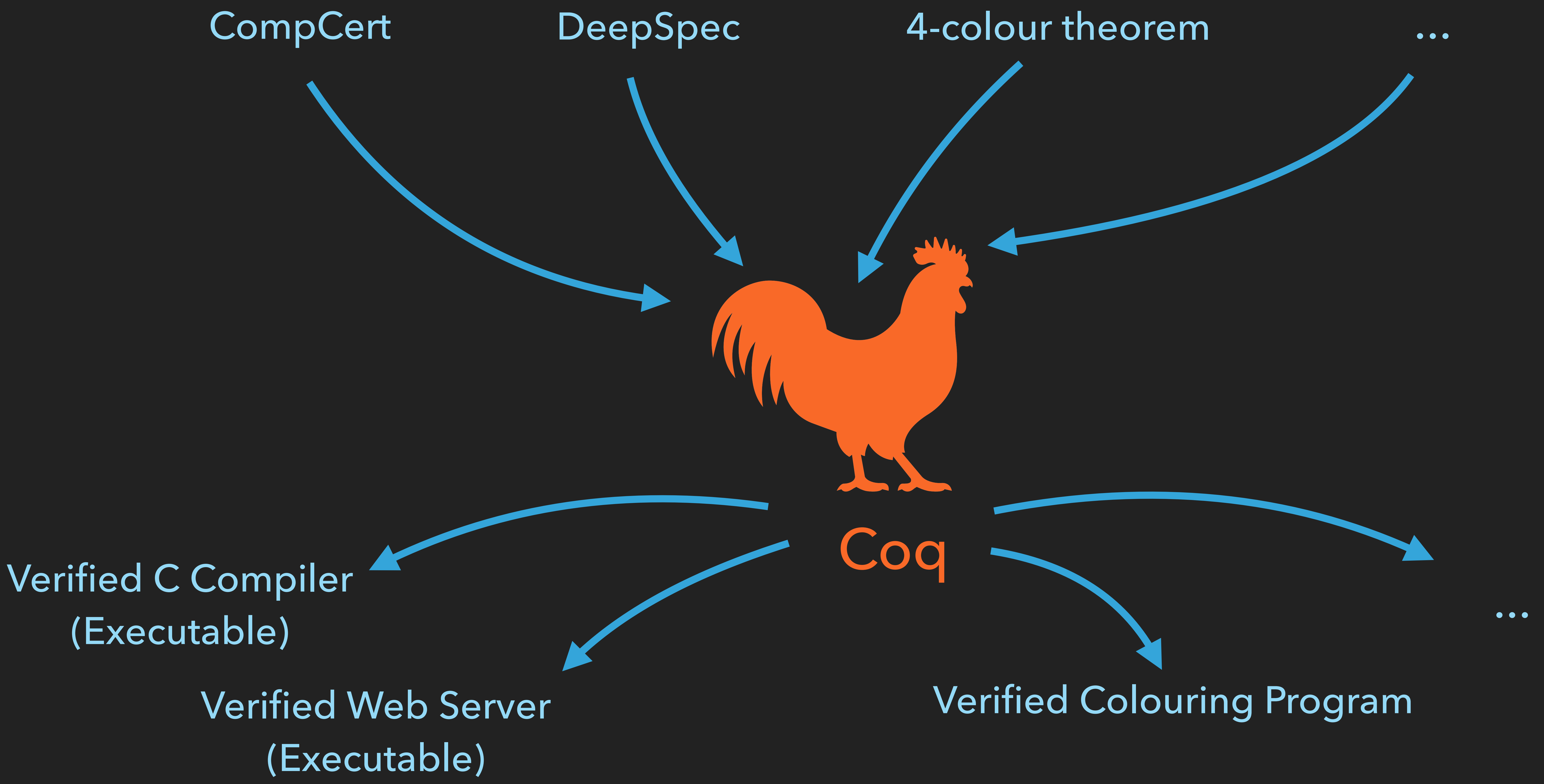
Matthieu **Sozeau**   Théo **Winterhalter**

joint work with

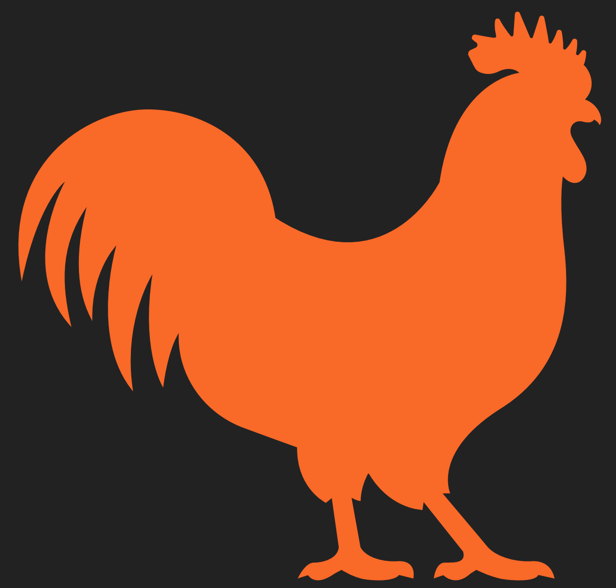
Jakob **Botsch Nielsen**   Simon **Boulier**

Yannick **Forster**   Nicolas **Tabareau**

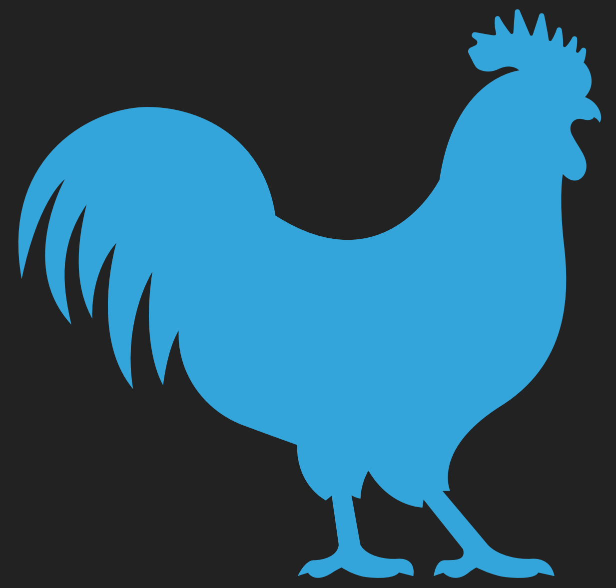
Motivation


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

What do you trust?

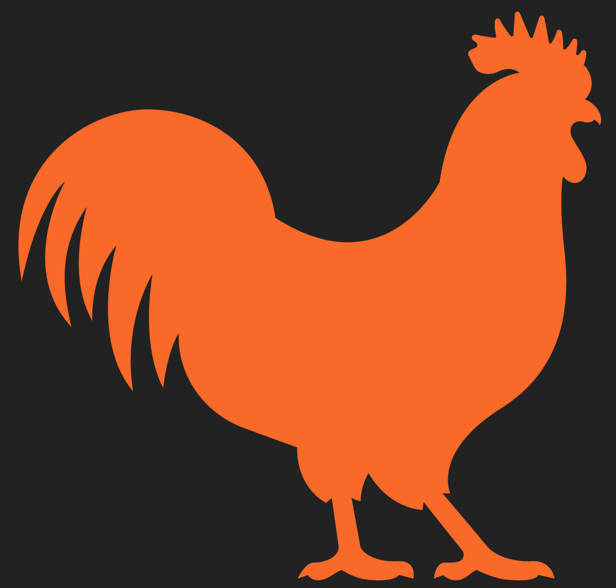


Ideal Coq

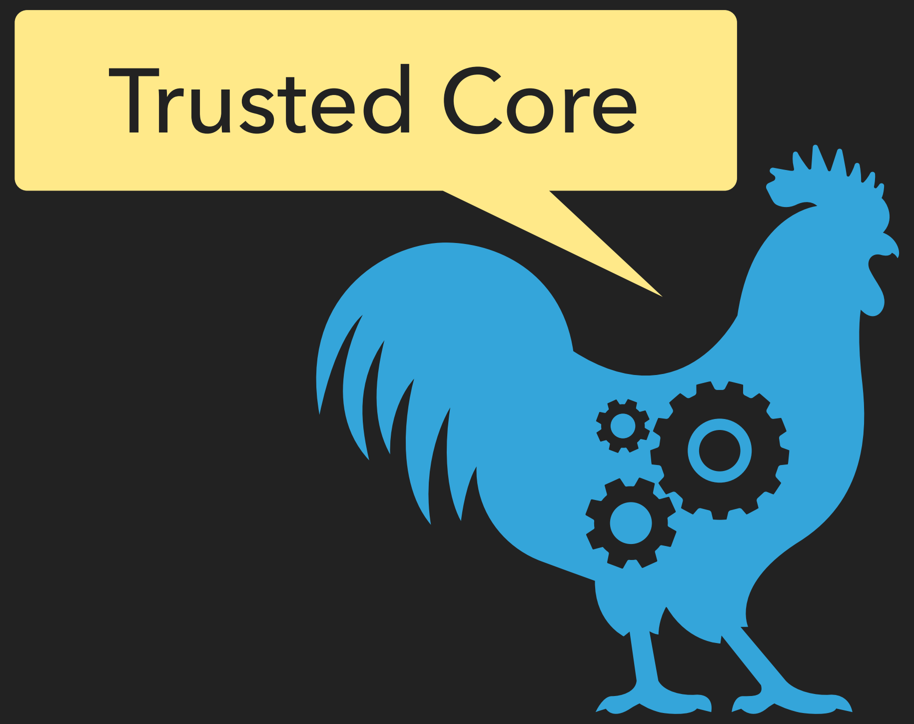


Implemented Coq

What do you trust?



Ideal Coq

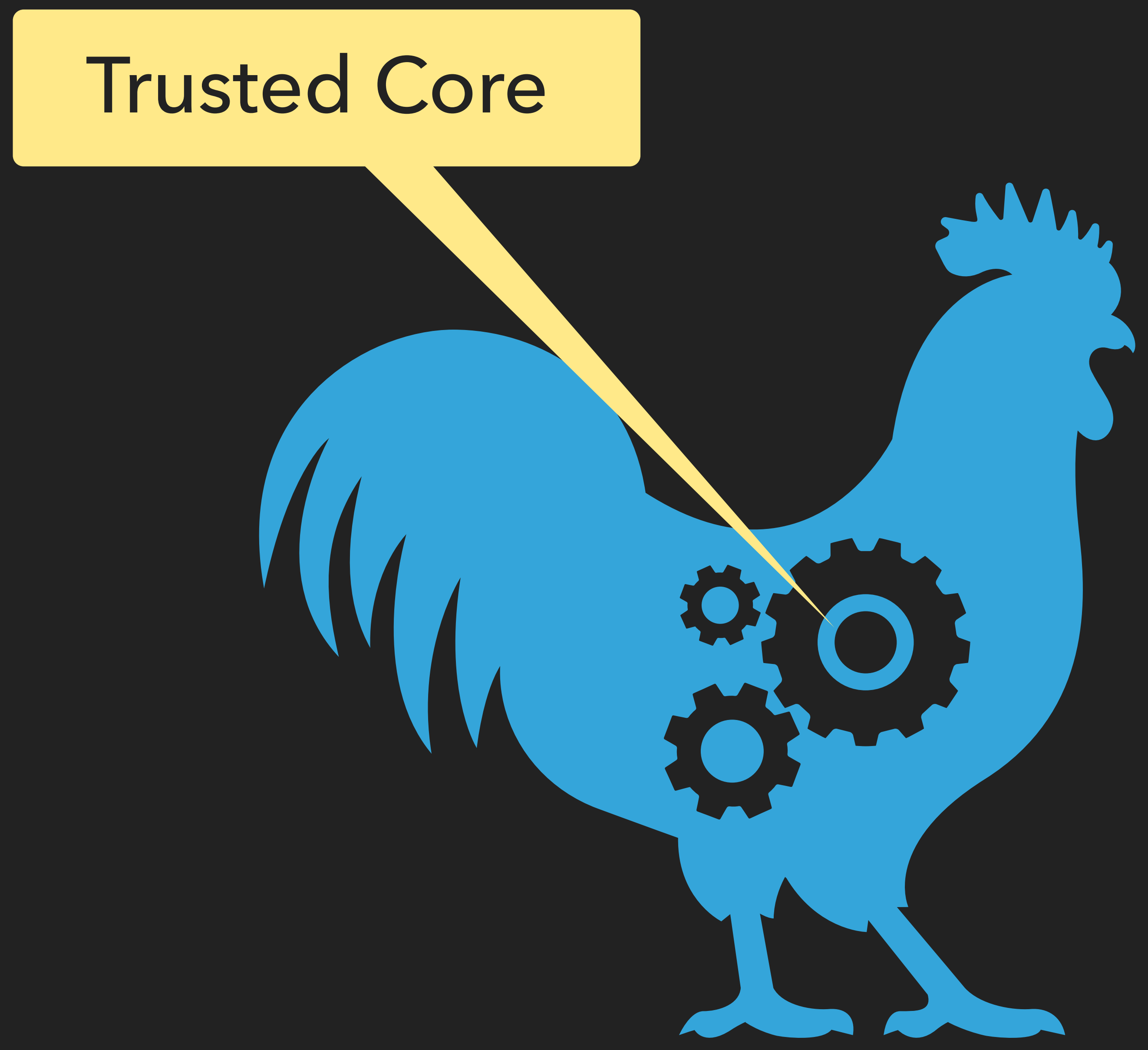


Implemented Coq

What do you trust?

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

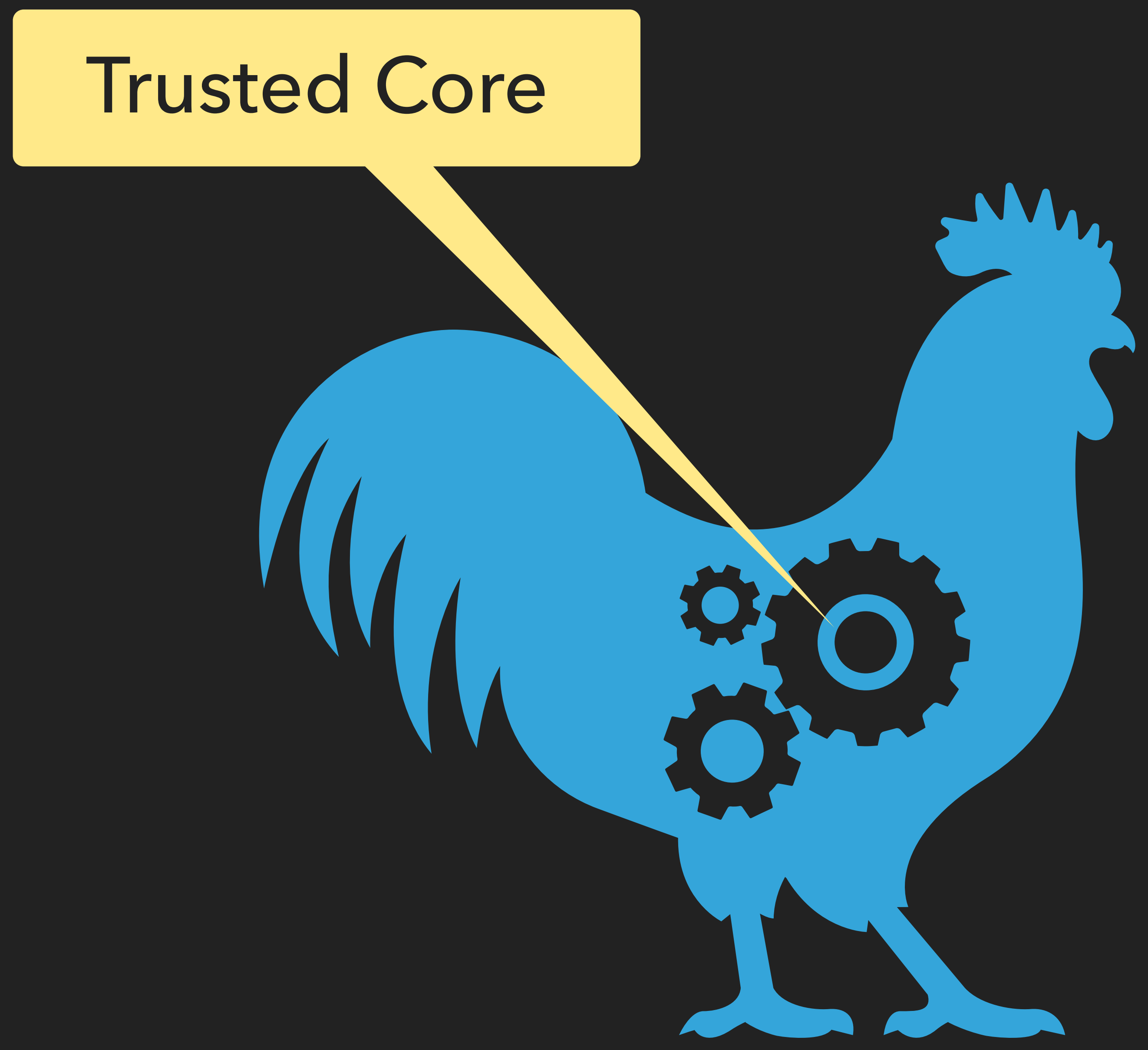


Implemented Coq

What do you trust?

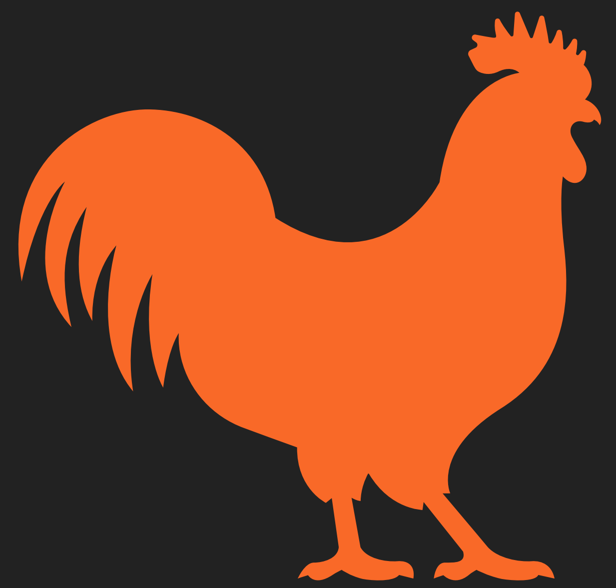
Dependent Type Checker (18kLoC, 30+ years)

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

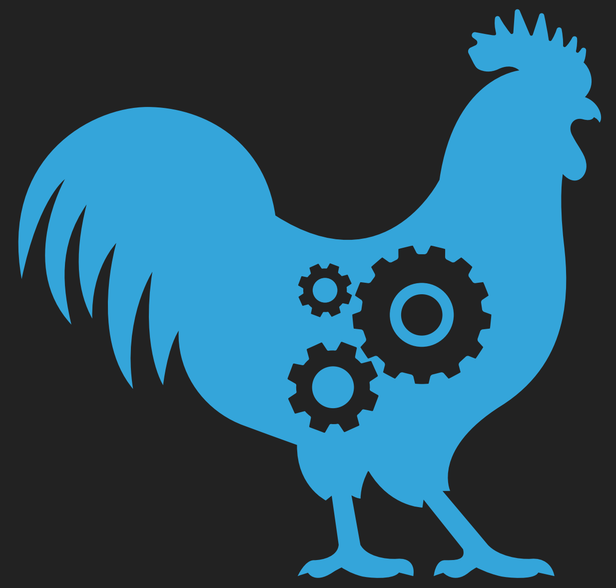


Implemented Coq

The Reality



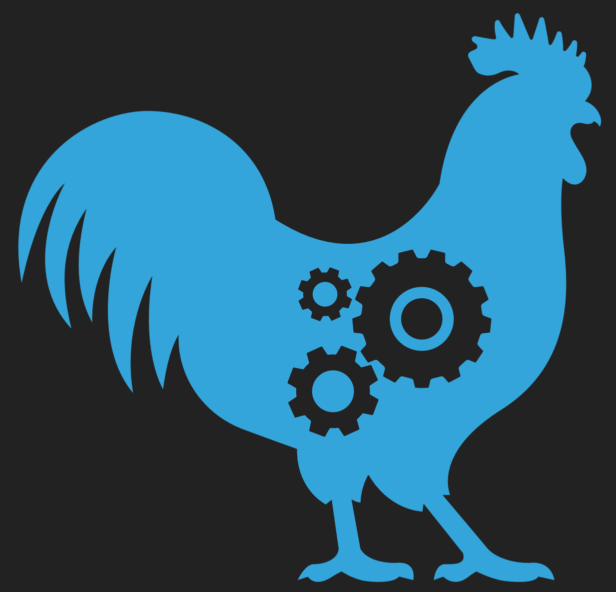
Ideal Coq



Implemented Coq

The Reality

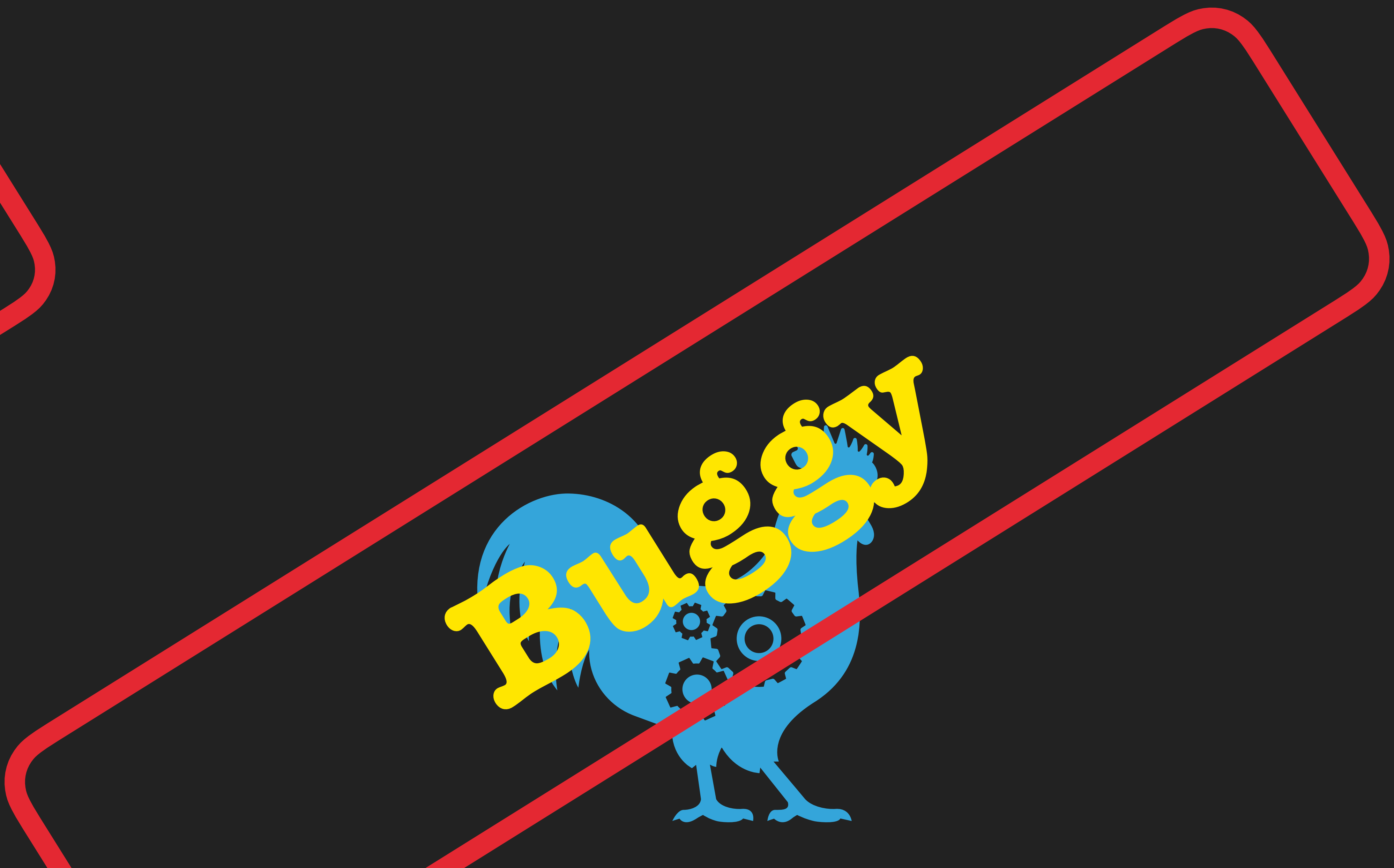
Ideal Coq



Implemented Coq

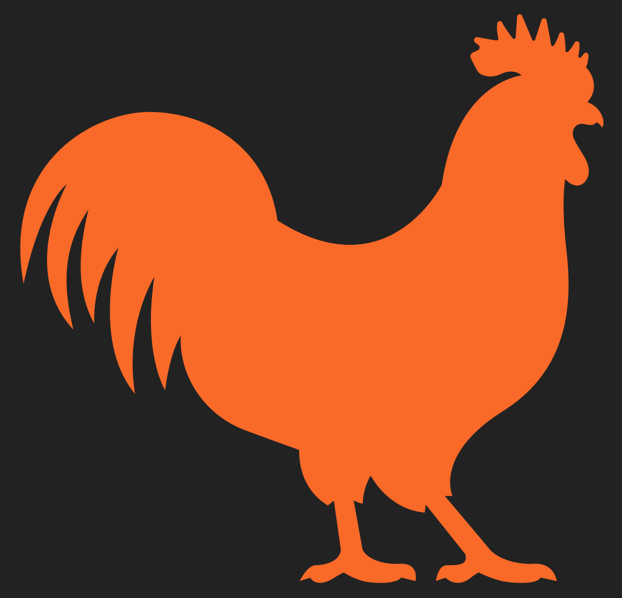
The Reality

Ideal Coq



Implemented Coq

The Reality



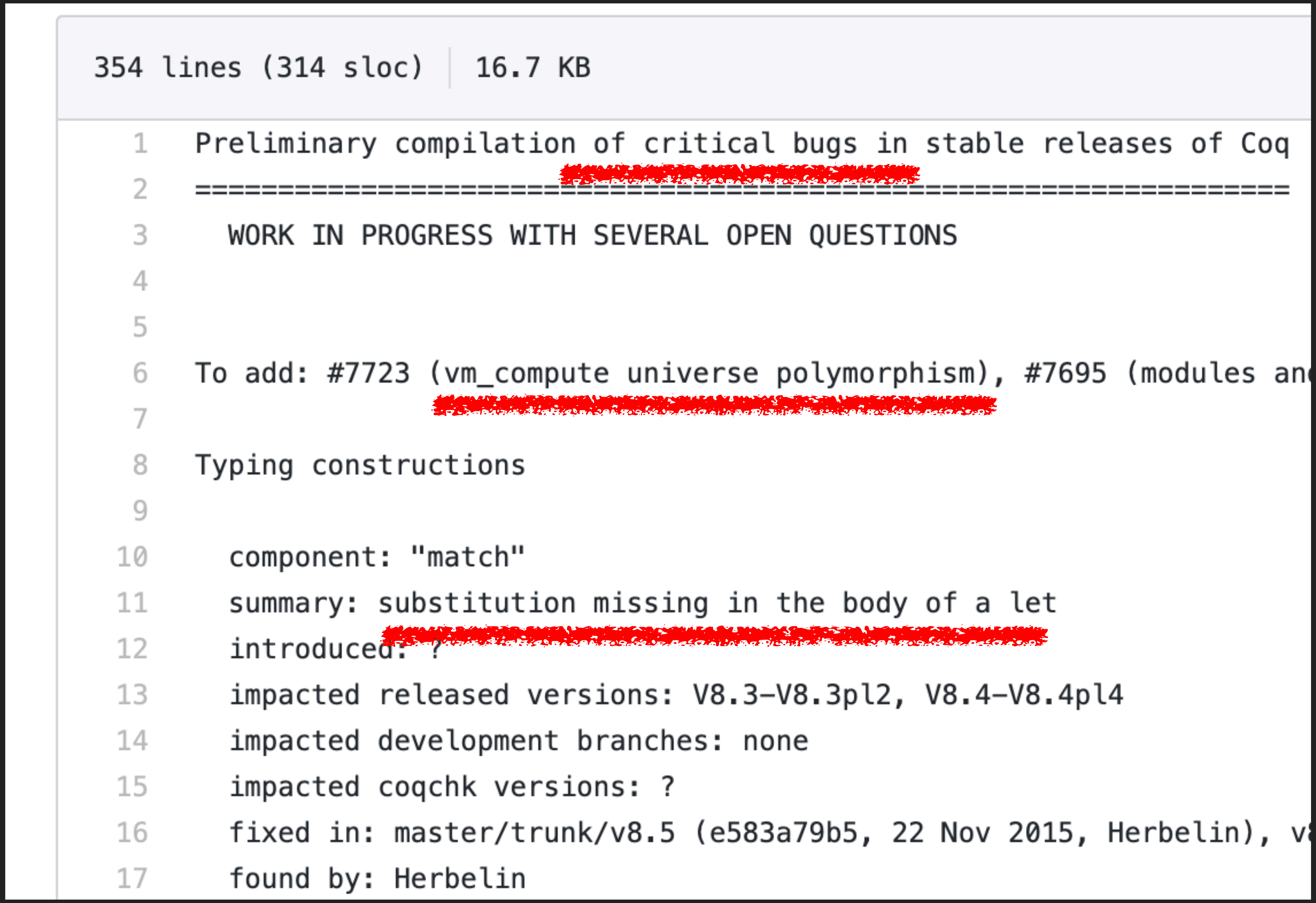
Ideal Coq

- Reference Manual roughly specifies on paper the basic core metatheory. The rest is (at best) in various papers and PhD theses, e.g. module system, treatment of eta-conversion, guard condition, SProp….
- Discrepancies with the actual implementation
- Combination of features not worked-out in detail. E.g. cumulative inductive types + let-bindings in parameters of inductives???

The Reality

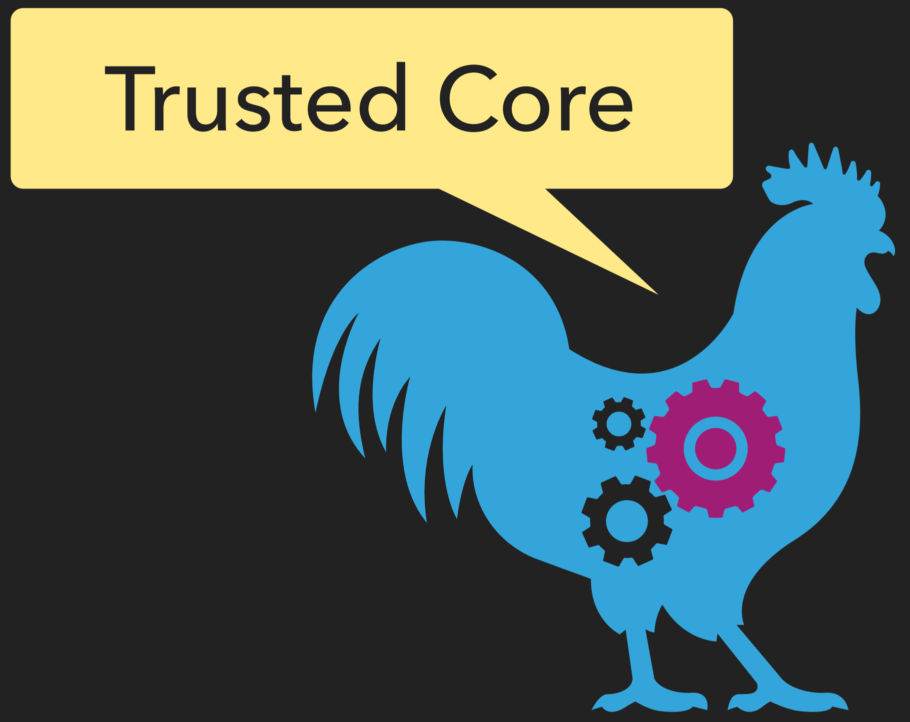
- Reference Manual roughly specifies on paper the basic core metatheory. The rest is (at best) in various papers and PhD theses, e.g. module system, treatment of eta-conversion, guard condition, SProp….
- Discrepancies with the actual implementation
- Combination of features not worked-out in detail. E.g. cumulative inductive types + let-bindings in parameters of inductives???

The Reality



354 lines (314 sloc) | 16.7 KB

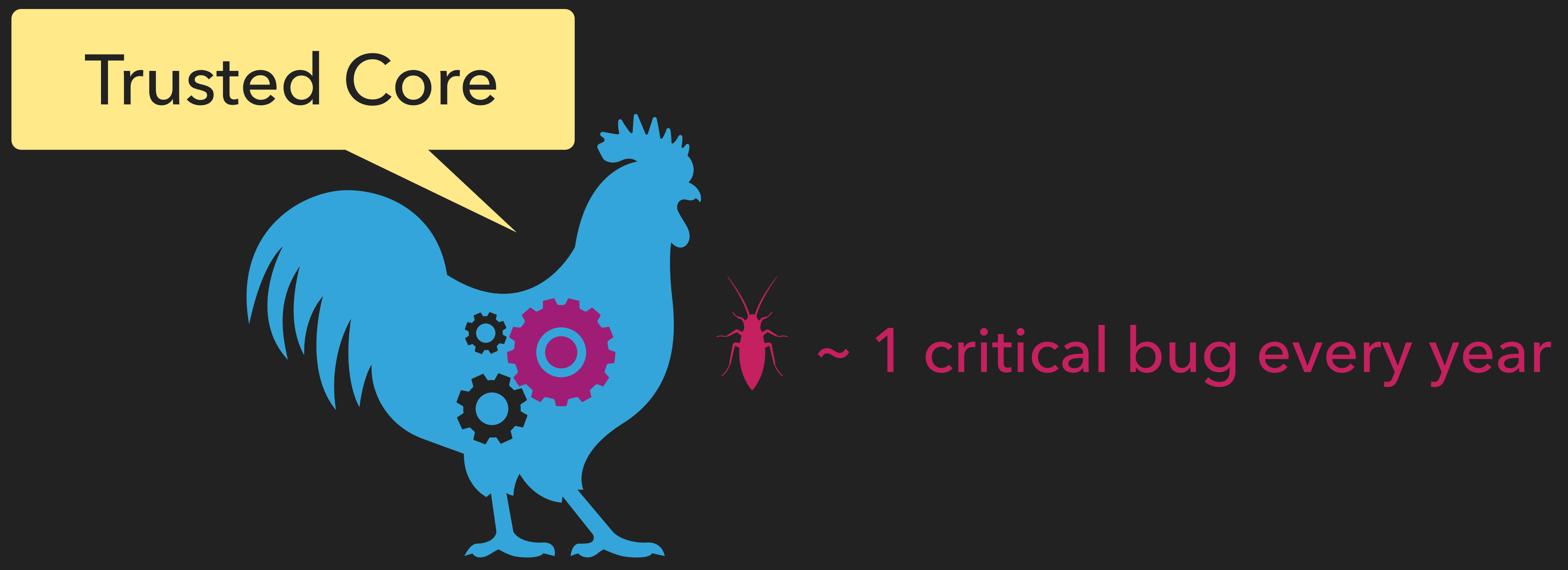
```
Preliminary compilation of critical bugs in stable releases of Coq
==================================================================
  WORK IN PROGRESS WITH SEVERAL OPEN QUESTIONS


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
```



Implemented Coq

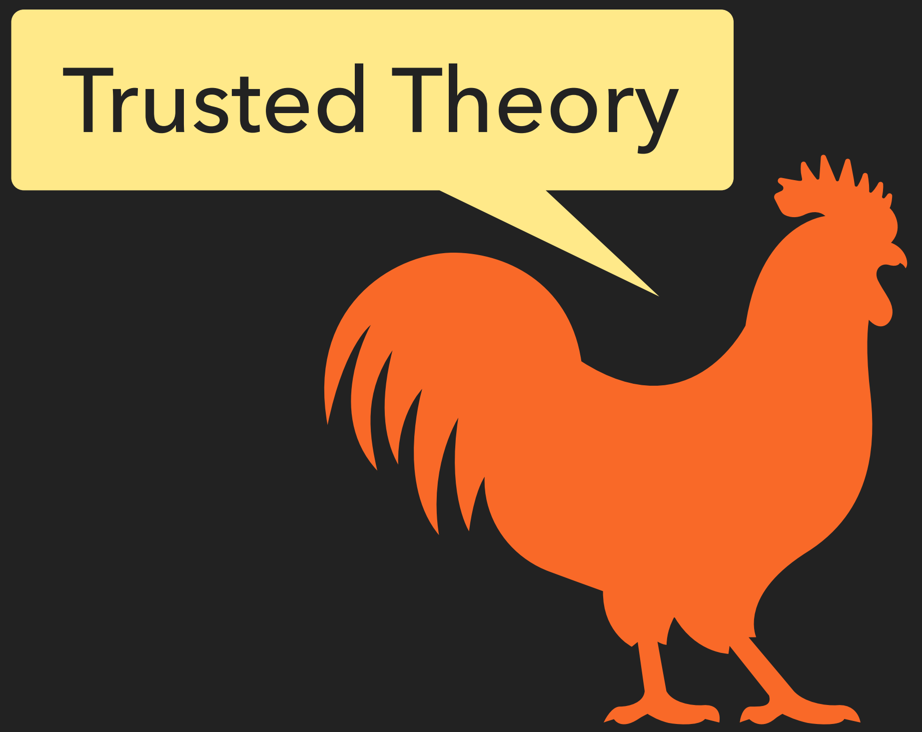
The Reality

```
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
```



Implemented Coq

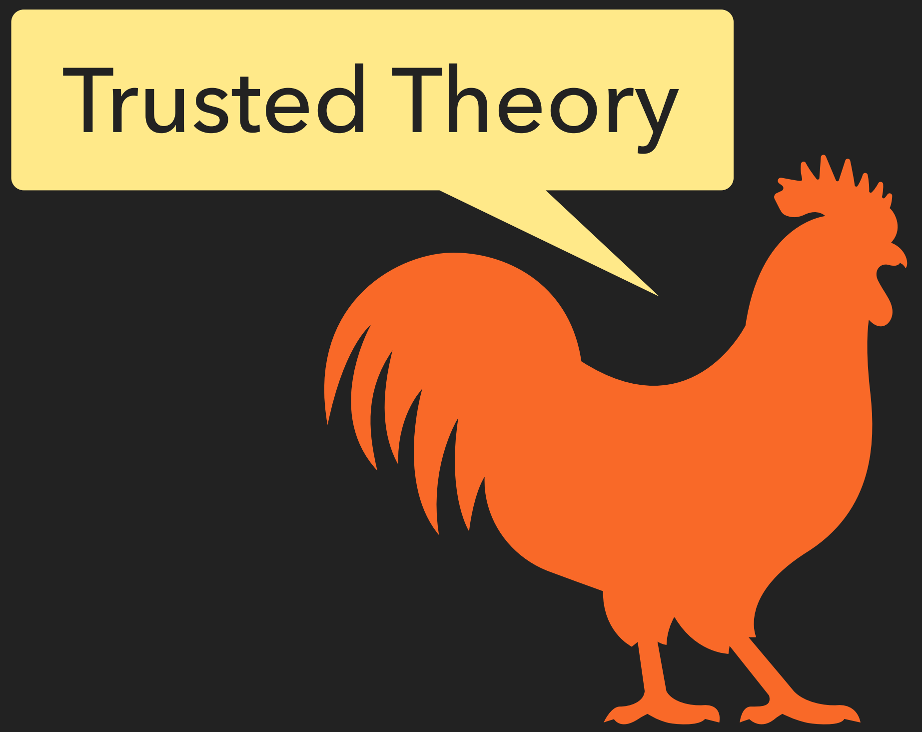
Our Goal: Improving Trust



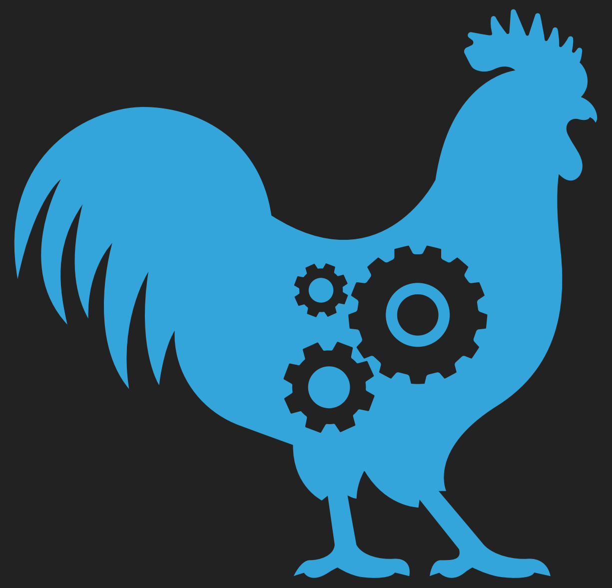
Ideal Coq

Implemented Coq

Our Goal: Improving Trust

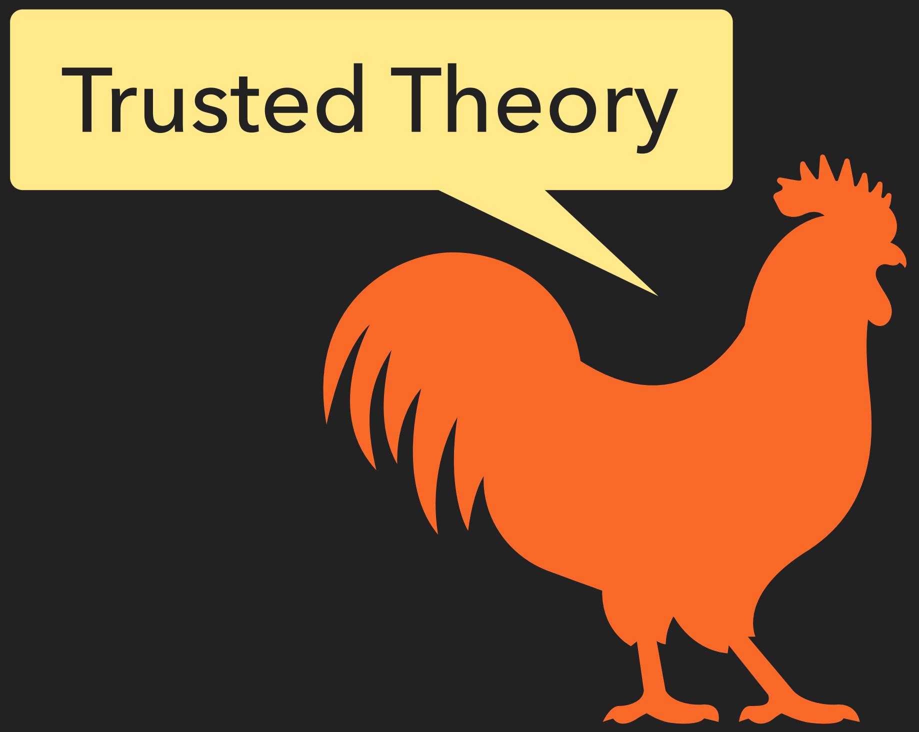


Ideal Coq



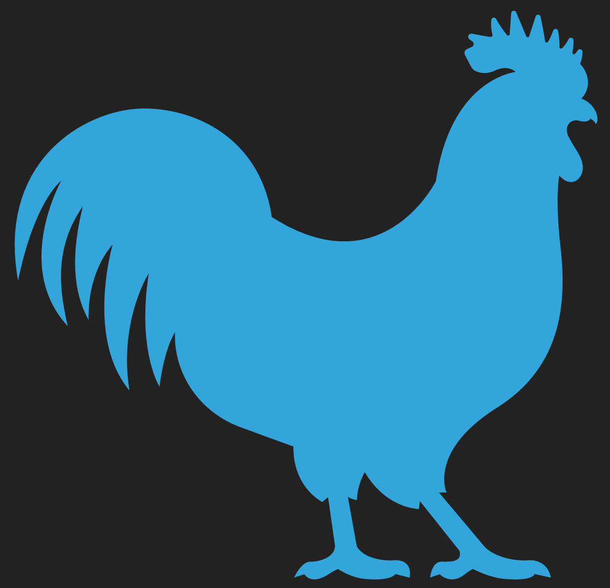
Implemented Coq

Coq in MetaCoq



Part I: Coq's Calculus PCUIC

↕

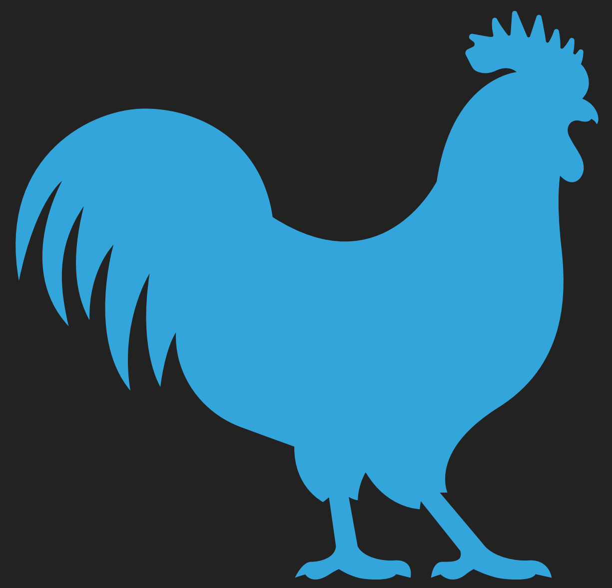


Part II: Verified Coq

in

MetaCoq
Formalization of
Coq in Coq
JAR'20

in



Implemented Coq

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.
```

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.
```

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.
```

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.
```

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.
```

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.
```

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 ∉ f)
```

η-conversion (WIP)

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

« template » polymorphism

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

module system

…and what we don't

```
(fun x ⇒ f x) = f   (x ∉ f)
```

η-conversion (WIP)

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

« template » polymorphism

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

module system

...and what we don't

```
(fun x ⇒ f x) = f   (x ∉ f)
```

η-conversion (WIP)

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

« template » polymorphism

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

module system

…and what we don't

```
(fun x ⇒ f x) = f   (x ∉ f)
```

η-conversion (WIP)

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

« template » polymorphism

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

module system

...and what we don't

```
(fun x ⇒ f x) = f   (x ∉ 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

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

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

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

Specification

Example: Reduction

DEFINITIONS IN CONTEXTS

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

GENERAL SUBSTITUTION

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

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

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

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

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

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

Specification

Example: Call-by-Value Evaluation

WEAK REDUCTION

CLOSED VALUE SUBSTITUTION

$$\frac{t \rightarrow_{cbv} v \qquad b[x := v] \rightarrow_{cbv} v'}{\texttt{let } x : T := t \texttt{ in } b \rightarrow_{cbv} v'}$$

$$\_ \rightarrow_{cbv} \_ \;\subseteq\; \varepsilon \vdash \_ \rightarrow \_$$

Meta-Theory

Structures

Meta-Theory

Structures

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

Meta-Theory

Structures

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

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

Meta-Theory

Structures

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

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
```

Meta-Theory

Structures

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

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

Meta-Theory

Judgments

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

One-step reduction and its reflexive transitive closure

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

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

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
wf Σ, wf_local Σ Γ	Well-formed global and local environments

Basic Meta-Theory

Structural Properties

Basic Meta-Theory

Structural Properties

- Traditional de Bruijn lifting and substitution operations in the spec

Basic Meta-Theory

Structural Properties

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

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

Basic Meta-Theory

Structural Properties

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

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

- Weakening and Substitution from renaming and instantiation theorems

Basic Meta-Theory

Structural Properties

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

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

- **Weakening** and **Substitution** from renaming and instantiation theorems
- Easier to lift to strengthening/exchange lemmas in the future (strengthening is not immediate here)

Universes

Universes

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

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 general than current Coq)

Universes

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

Typing $\Sigma\ ;\ \Gamma \vdash$ tSort u : tSort (Universe.super u)

No distinction of *algebraic* universes (more general than current Coq)

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

Universes

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

Typing $\Sigma\ ;\ \Gamma \vdash$ `tSort u : tSort (Universe.super u)`

No distinction of *algebraic* universes (more general than current Coq)

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

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

Universes

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

Typing $\Sigma\ ;\ \Gamma \vdash$ `tSort u : tSort (Universe.super u)`

No distinction of *algebraic* universes (more general than current Coq)

```
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.

Universes

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

Typing $\Sigma \,;\, \Gamma \vdash$ `tSort u : tSort (Universe.super u)`

No distinction of *algebraic* universes (more general than current Coq)

```
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

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)

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`

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

Meta-Theory

The path to subject reduction

Validity

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

Meta-Theory

The path to subject reduction

Validity

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

Context Conversion

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

Meta-Theory

The path to subject reduction

Validity

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

Context Conversion

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

Subject Reduction

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

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

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

Subject Reduction

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

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**

Subject Reduction

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

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**

Subject Reduction

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

Relies on injectivity of product types, a consequence of **confluence**

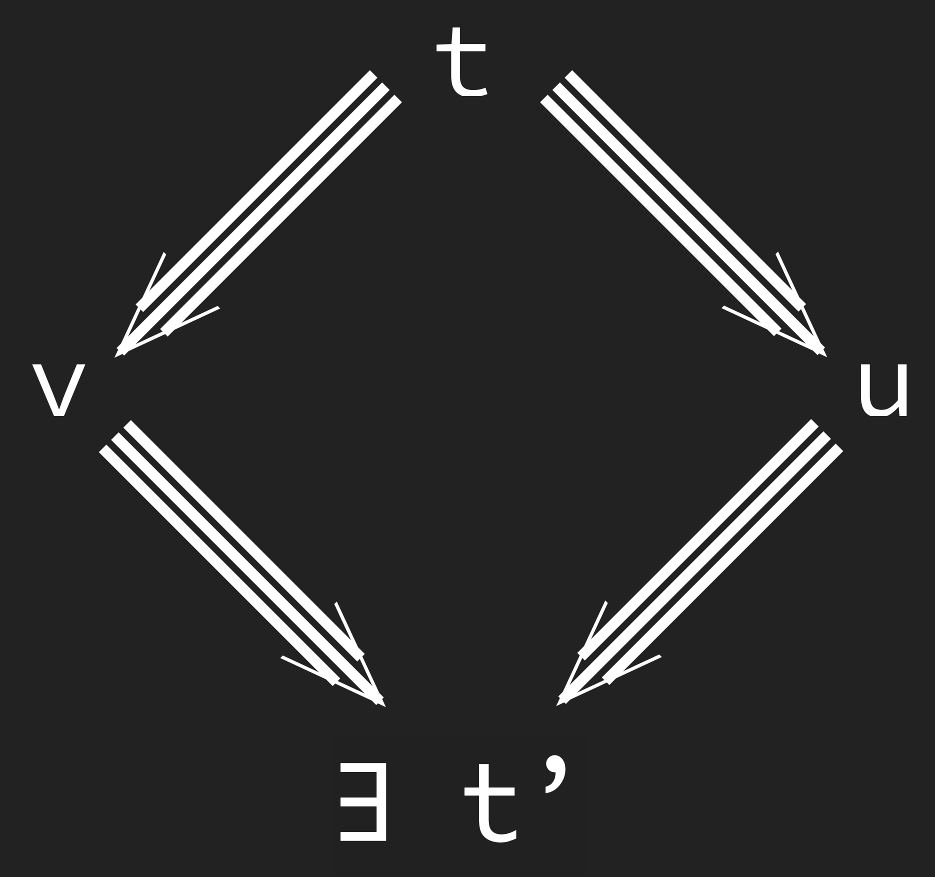
Confluence

The traditional way

Σ , Γ ⊢ t ⇛ u   One-step parallel reduction

À la Tait-Martin-Löf/Takahashi:

Diamond for ⇛



“Squash” lemma

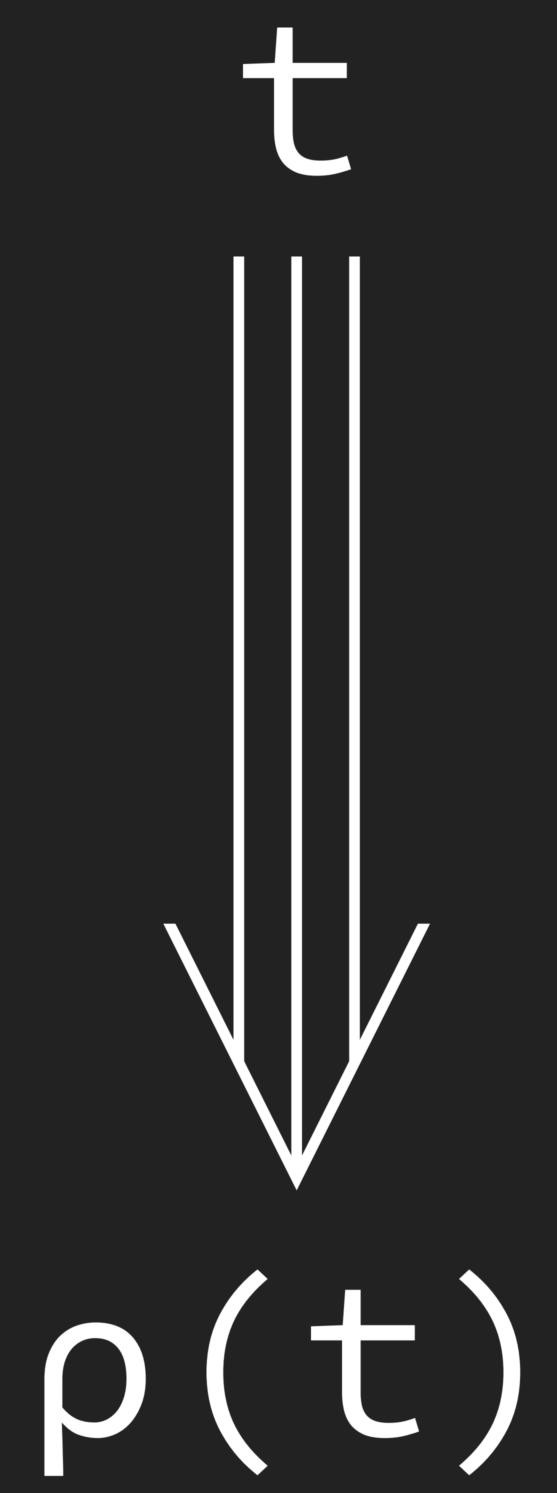
_ → _

⊂ _ ⇛ _

⊂ _ →* _

Takahashi’s Trick

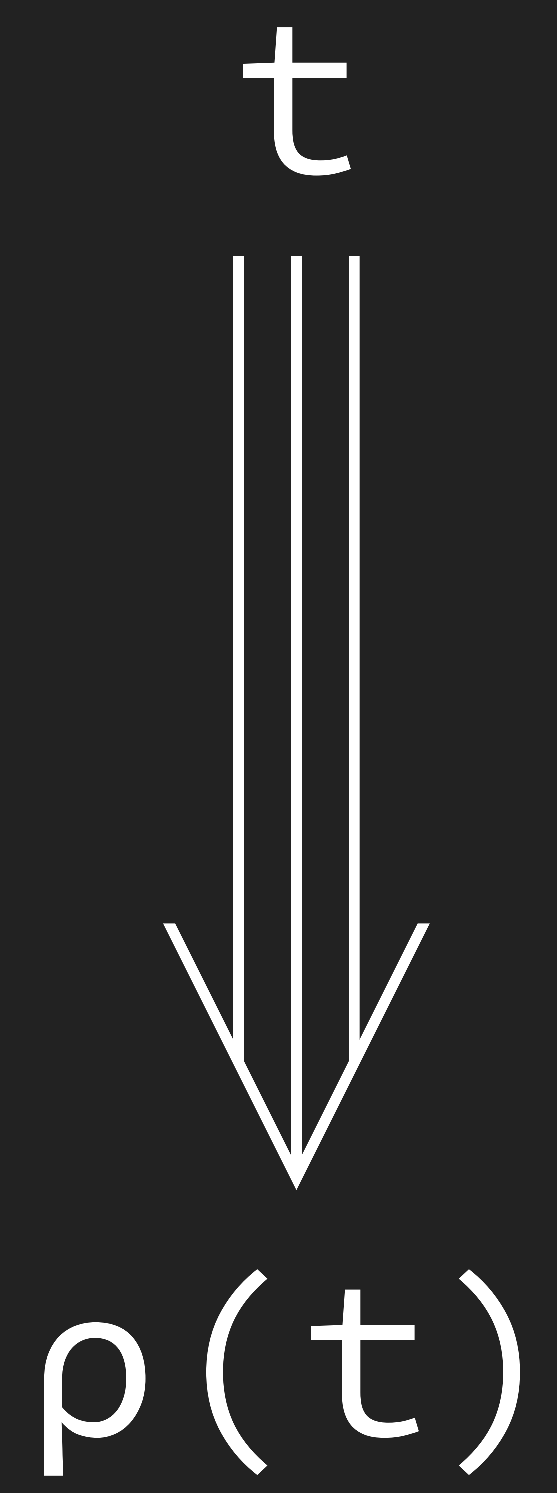
```
ρ : term -> term
```



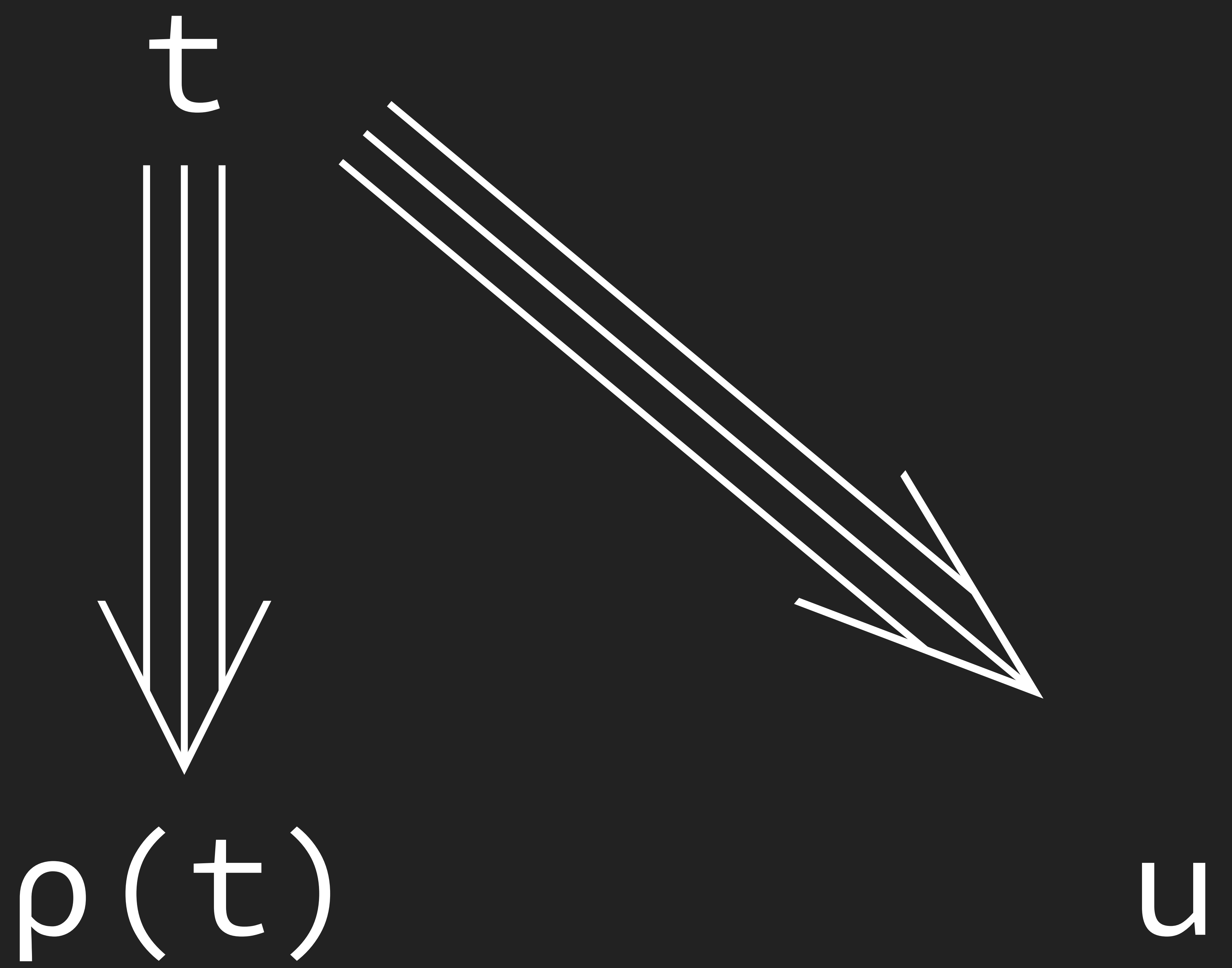
Takahashi's Trick

```
ρ : term -> term
```

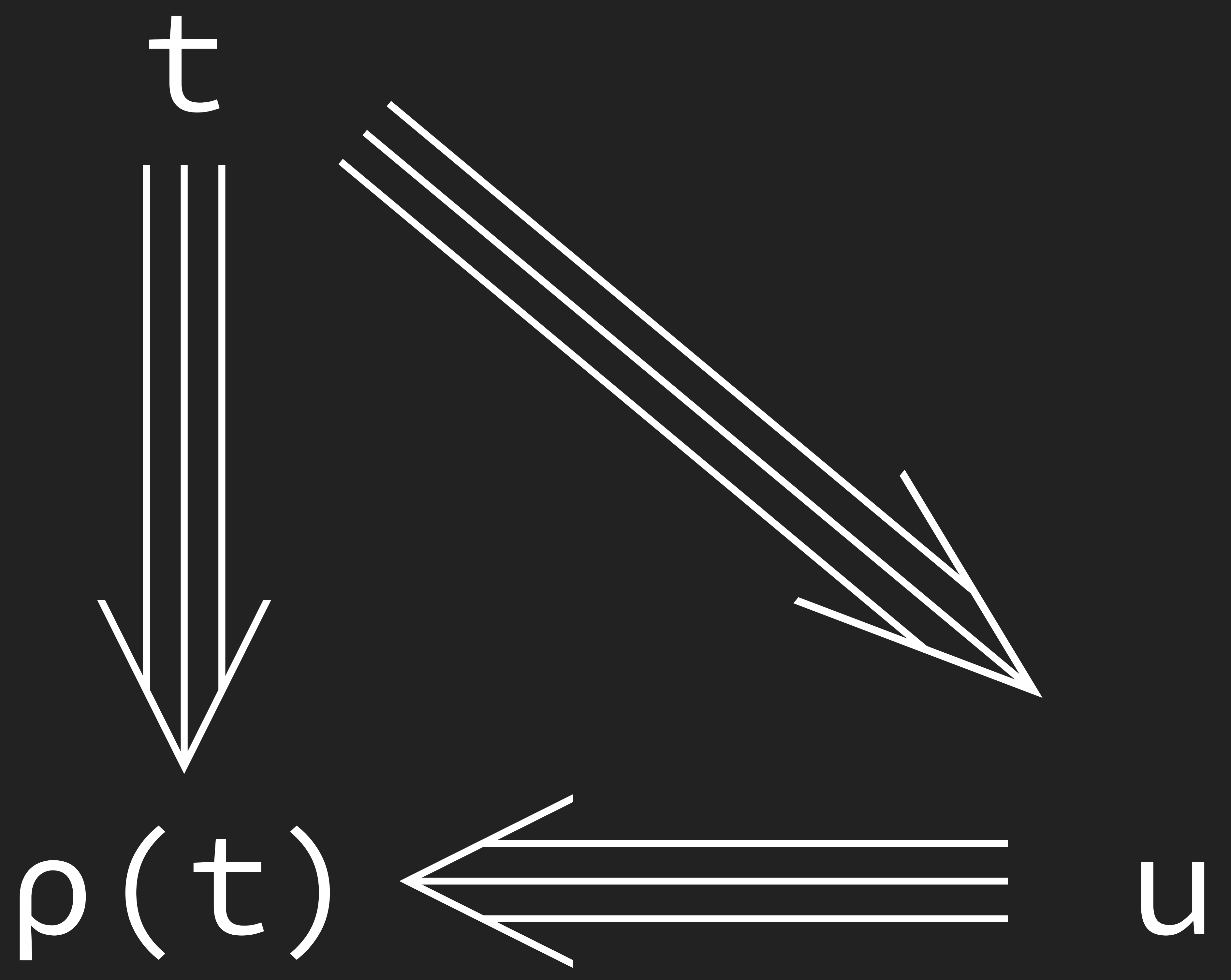
An *optimal* one-step parallel reduction function.



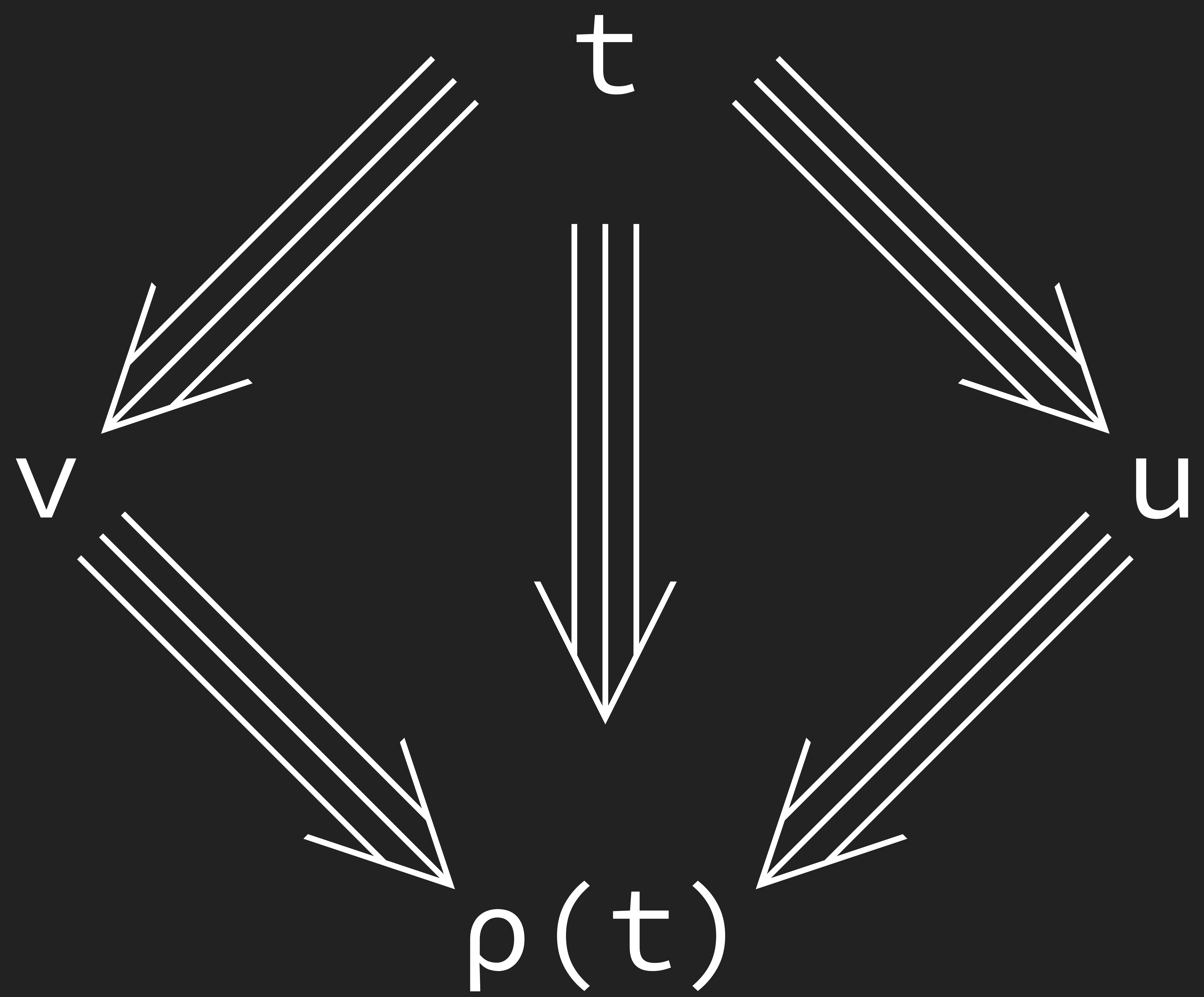
The triangle property



The triangle property



The triangle property



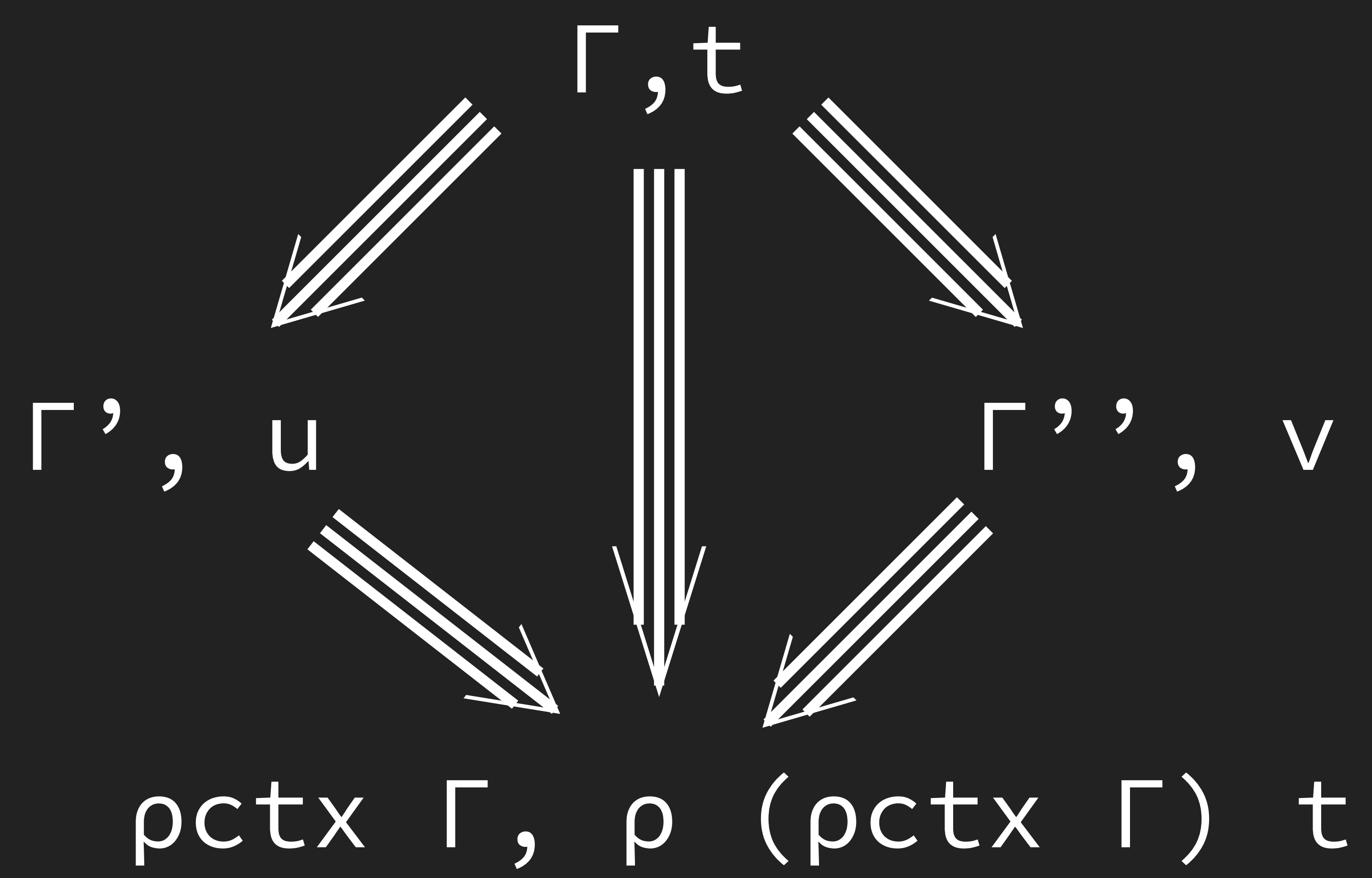
Confluence

For a theory with definitions in contexts

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

One-step parallel reduction, including reduction in contexts.

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



Confluence

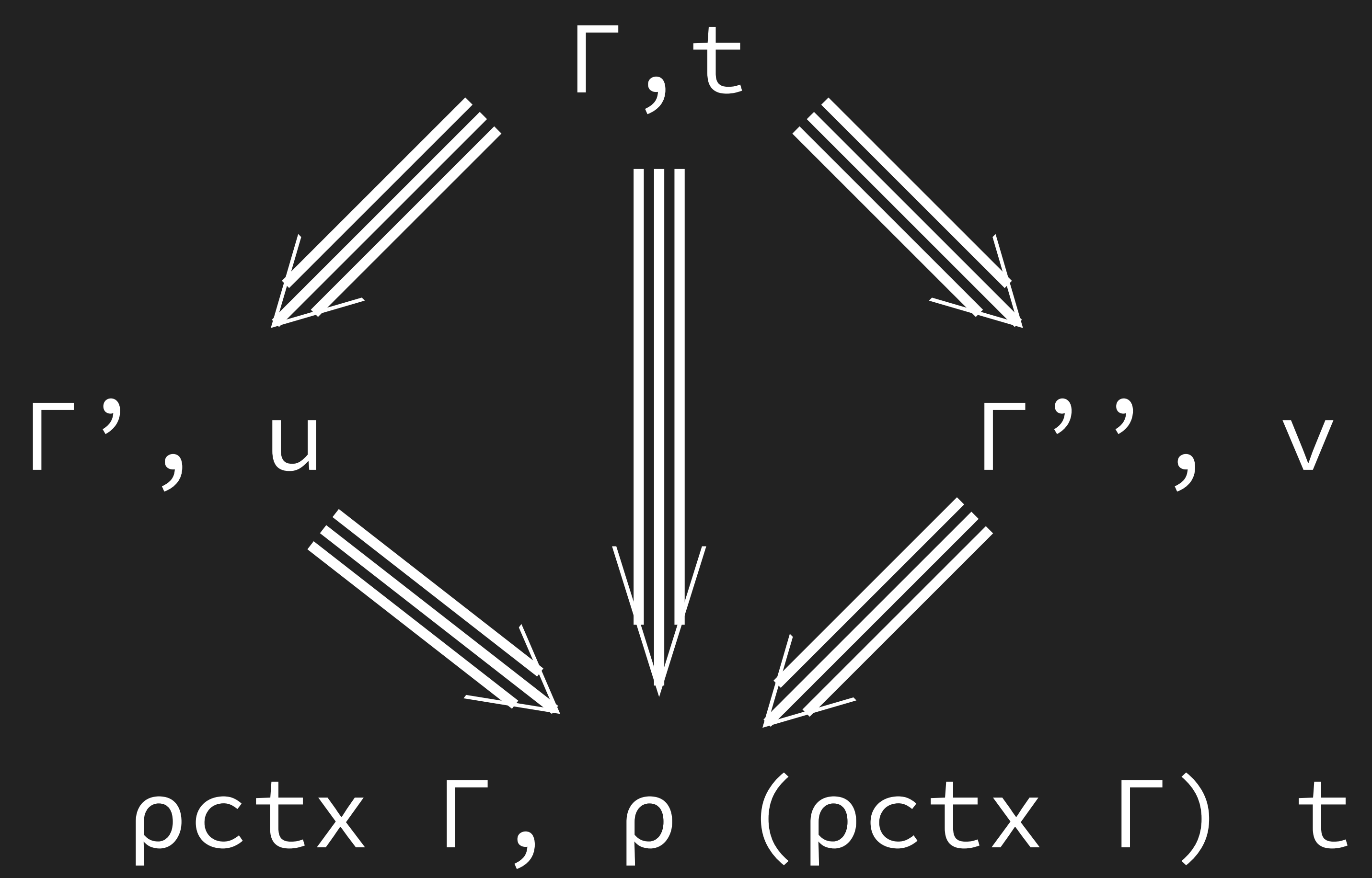
For a theory with definitions in contexts

```
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
```



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
```

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

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

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

Cumulativity and Prop

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

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

Trusted Theory Base

Assumptions

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch (verified and testable)

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch (verified and testable)

- Guard Conditions. Oracles:
  `check_fix : global_env → context → fixpoint → bool`
  + preservation by renaming/instantiation/equality/reduction

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch (verified and testable)

- Guard Conditions. Oracles:
  `check_fix : global_env → context → fixpoint → bool`
  + preservation by renaming/instantiation/equality/reduction

- Strong Normalization (not provable thanks to Gödel, but also not used in the preceding results). Consistency and canonicity follow easily.

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

Verifying Type-Checking

Conversion

Objective

Conversion

Objective

Input

u : A

v : B

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

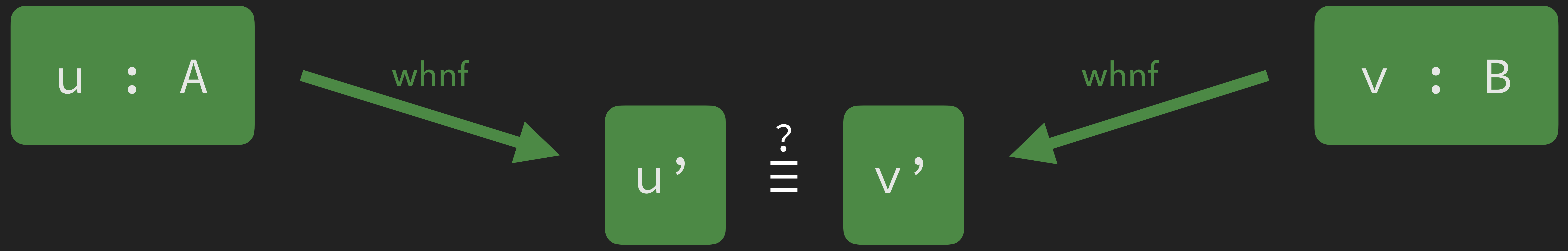
v : B

Conversion

Algorithm

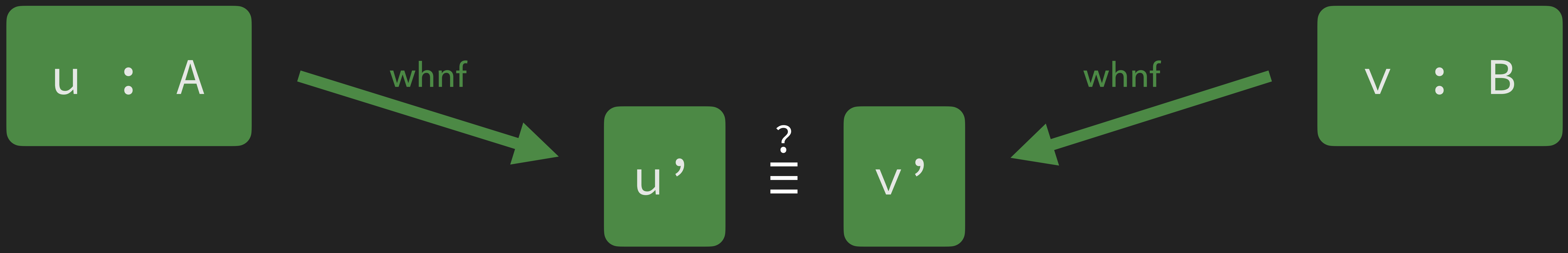
Conversion

Algorithm



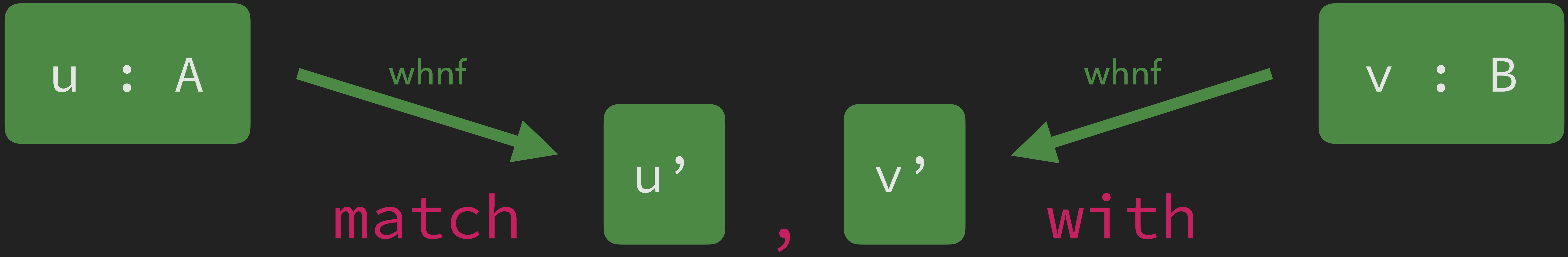
Conversion

Algorithm



Conversion

Algorithm



Conversion

Algorithm

u : A —whnf→ u'

v : B —whnf→ v'

```
match u' , v' with
λ(x:A₁). t₁ , λ(x:A₂). t₂
```

Conversion

Algorithm

u : A —whnf→ u'

v : B —whnf→ v'

match u' , v' with

λ(x:A₁). t₁ , λ(x:A₂). t₂ ⇒

Conversion

Algorithm

u : A —whnf→ u'

v : B —whnf→ v'

match u' , v' with

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

Conversion

Algorithm

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

`match` u' , v' `with`

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

Conversion

Algorithm

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

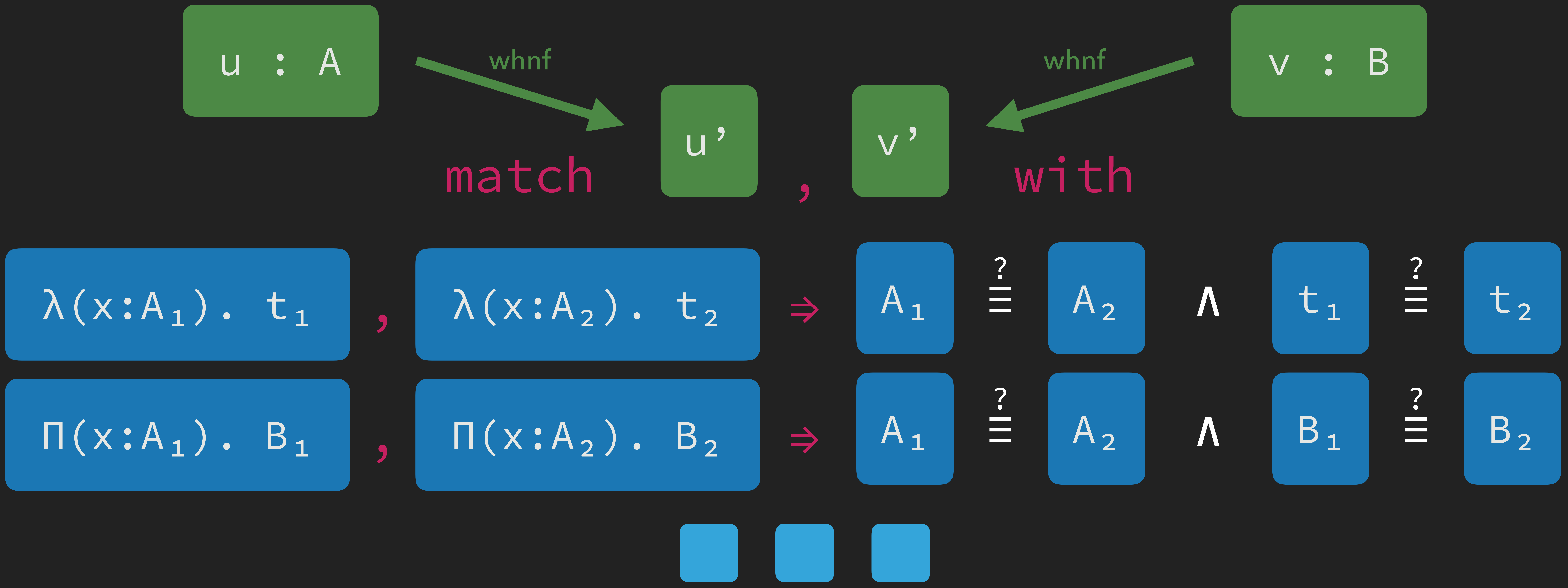
match u' , v' with

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

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

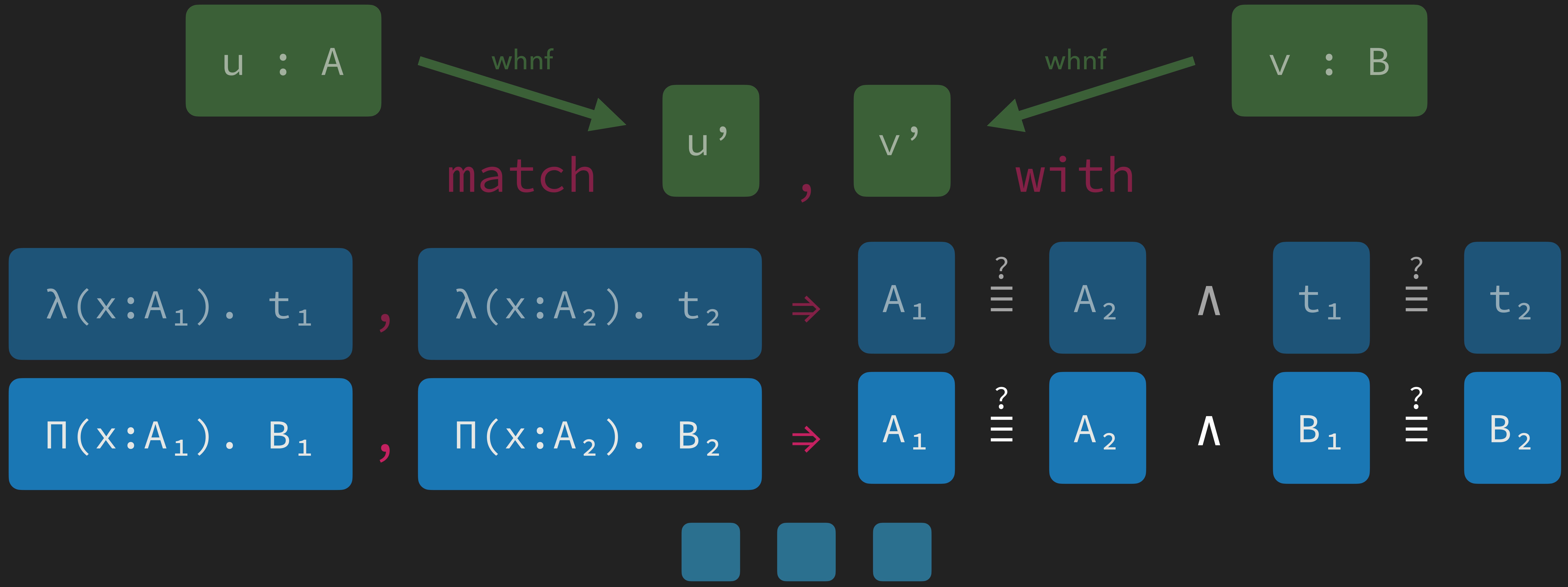
Conversion

Algorithm



Conversion

Completeness



Conversion

Completeness

$u : A$ —whnf→ u' ; $v : B$ —whnf→ v'

`match` u' , v' `with`

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

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

Conversion

Completeness

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

Conversion

Completeness

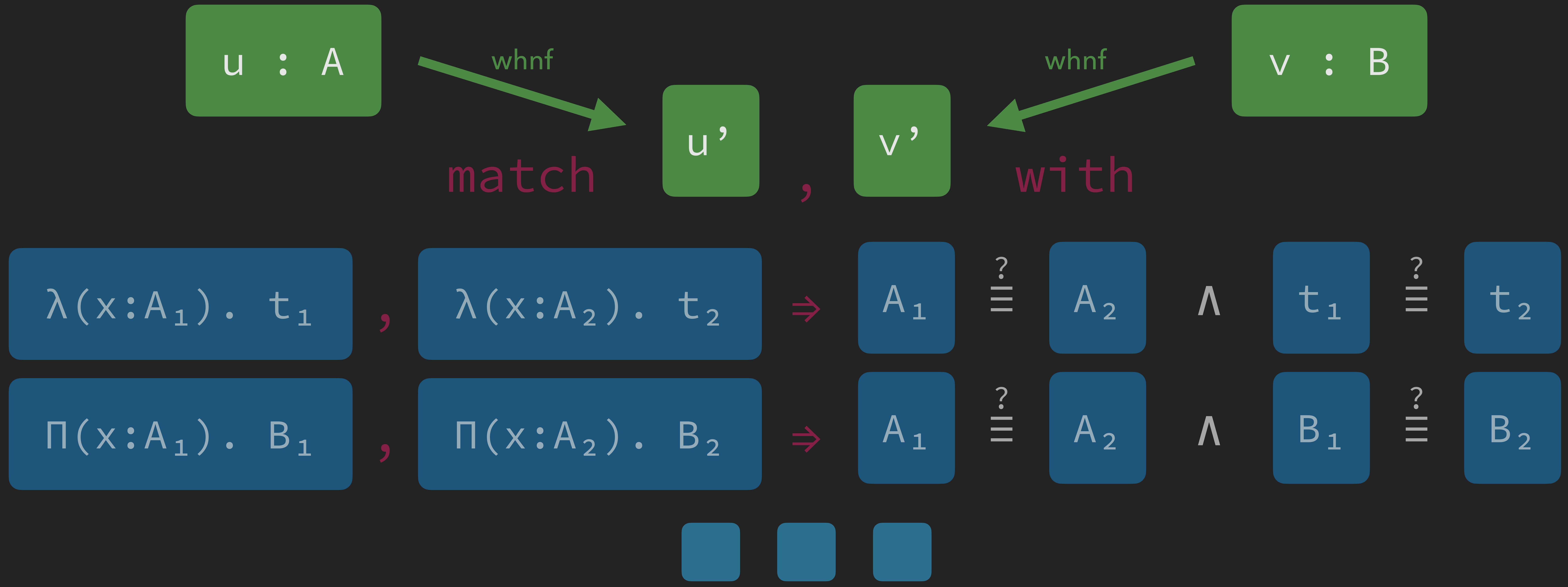
$\Pi(x{:}A_1).\ B_1 \overset{?}{\equiv} \Pi(x{:}A_2).\ B_2 \Rightarrow A_1 \not\equiv A_2$

we conclude

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

using inversion lemmata and confluence

Conversion


u : A
whnf
u'
v : B
whnf
v'
match u' , v' with
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

Weak head reduction

Objective

Input

u

Weak head reduction

Objective

Input

u

Output

v

Weak head reduction

Objective

Input	Output
u	v
`term`	`term`

Weak head reduction

Objective

Input

u
`term`

Output

v	u → v
`term`	`Prop`

Weak head reduction

Objective

Input

u

term

Output

v | u → v

term | Prop

```
weak_head_reduce : ∀ (u : term), ∑ (v : term), u → v
```

Weak head reduction

Example

Input u

Output v u → v

```
foo 0
```

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
```

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

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

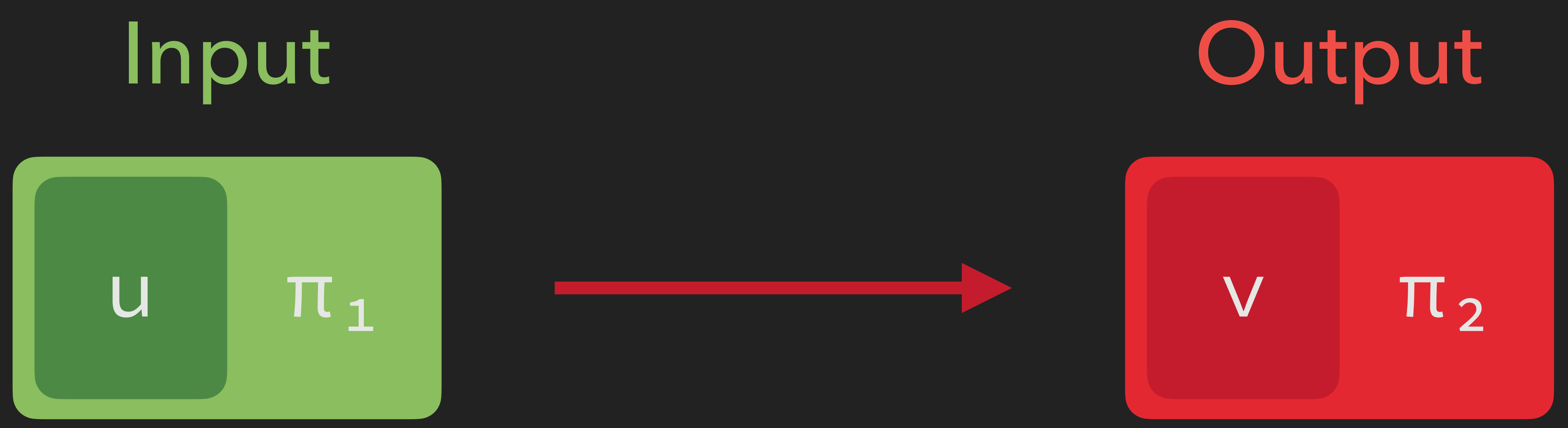
Output

v π_2

$u \to v$

Weak head reduction

Termination



Weak head reduction

Termination

```
foo 0
```

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`

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` ⟶ `(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 $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

`p.1`

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

`p.1`

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

p . 1

1 Lexicographic order of ← and ⊏

Weak head reduction

Termination

`p.1` `p.1`

Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

`p.1` `p.1`

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

`p.1`   `p.1`

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

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

`p.1` > `p.1`

and `p.1 = p.1`

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

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

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

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

1 Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

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

1 Lexicographic order of $\leftarrow$ 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
```

▼

```
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 $\prec$ and $\sqsubset$~~

Weak head reduction

Termination

Lexicographic order of $\leftarrow$ and $\sqsubset$

Weak head reduction

Termination

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

Weak head reduction

Termination

2 Lexicographic order of ← and an order on positions

Weak head reduction

Termination

Lexicographic order of ← and an order on positions

Weak head reduction

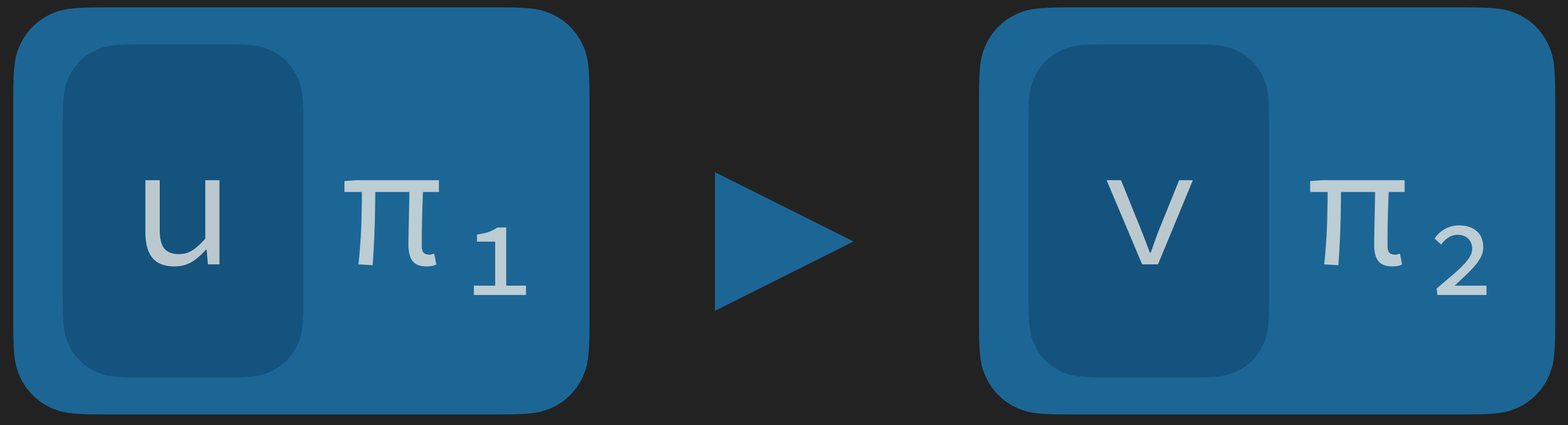
Termination

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

Lexicographic order of $\leftarrow$ and an order on positions

Weak head reduction

Termination



$$\langle\ u\ \pi_1\ ,\ \mathtt{stack\_pos}\ u\ \pi_1\ \rangle > \langle\ v\ \pi_2\ ,\ \mathtt{stack\_pos}\ v\ \pi_2\ \rangle$$

Lexicographic order of ← and an order on positions

Weak head reduction

Termination

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

$\langle \ u \ \pi_1 \ , \ \text{stack\_pos} \ u \ \pi_1 \ \rangle > \langle \ v \ \pi_2 \ , \ \text{stack\_pos} \ v \ \pi_2 \ \rangle$

$\text{pos} \ (u \ \pi_1)$

Lexicographic order of ← and an order on positions

Weak head reduction

Termination

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

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

2 Lexicographic order of ← and an order on positions

Weak head reduction

Termination

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

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

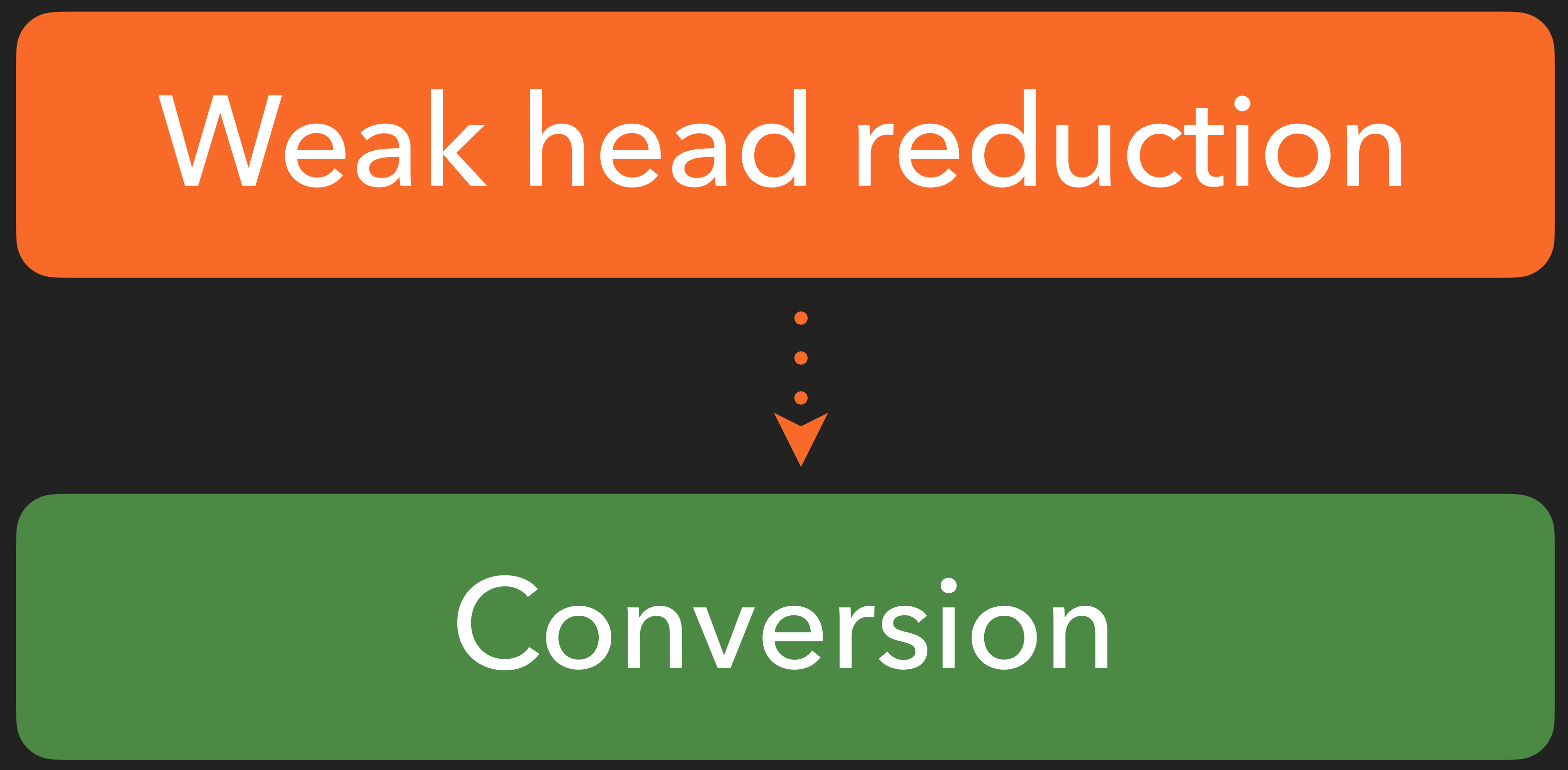
Dependent lexicographic order of ← and an order on positions

Type Checking

Type Checking

Weak head reduction

Type Checking



Type Checking

Weak head reduction

↓

Cumulativity

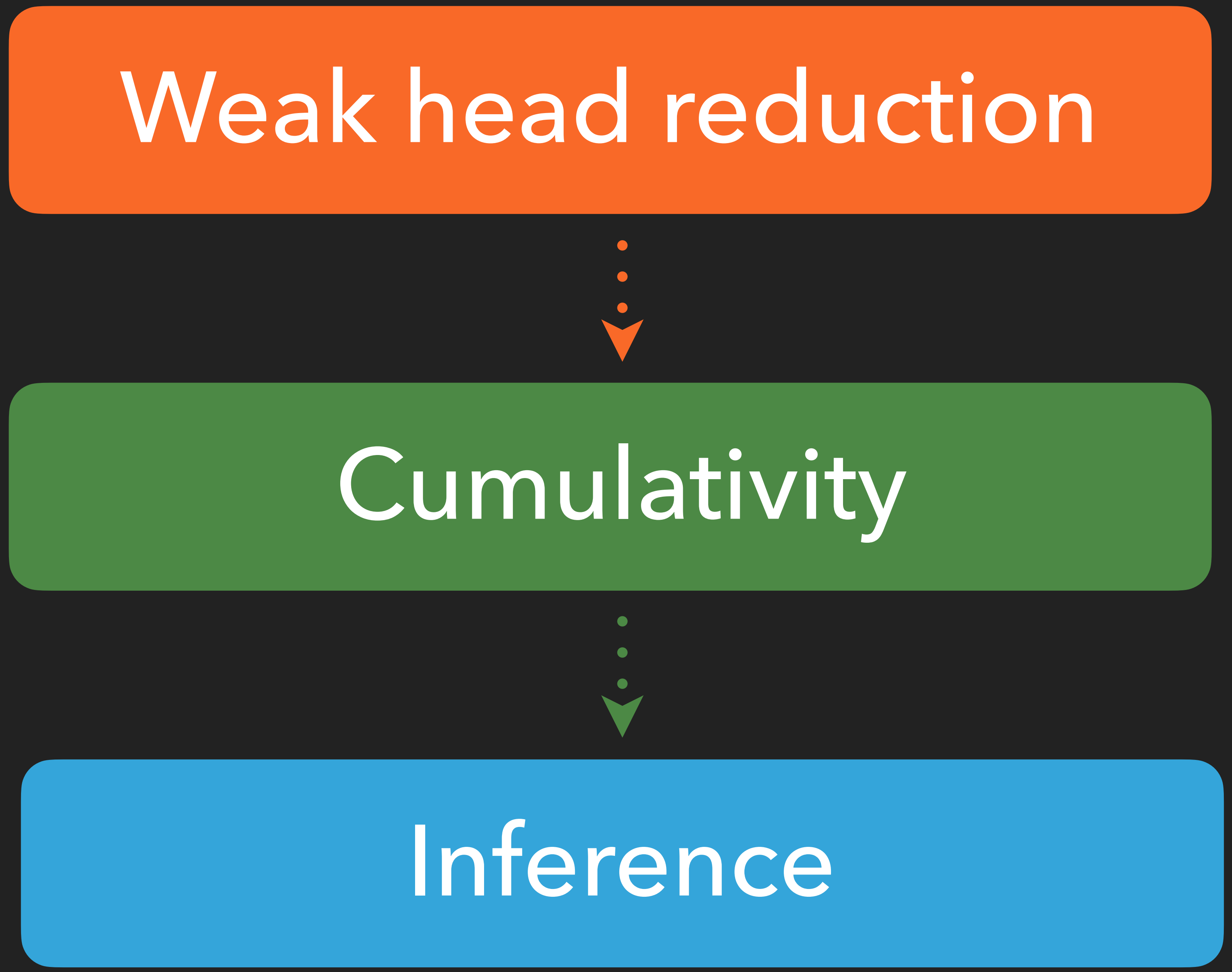
Type Checking

Weak head reduction

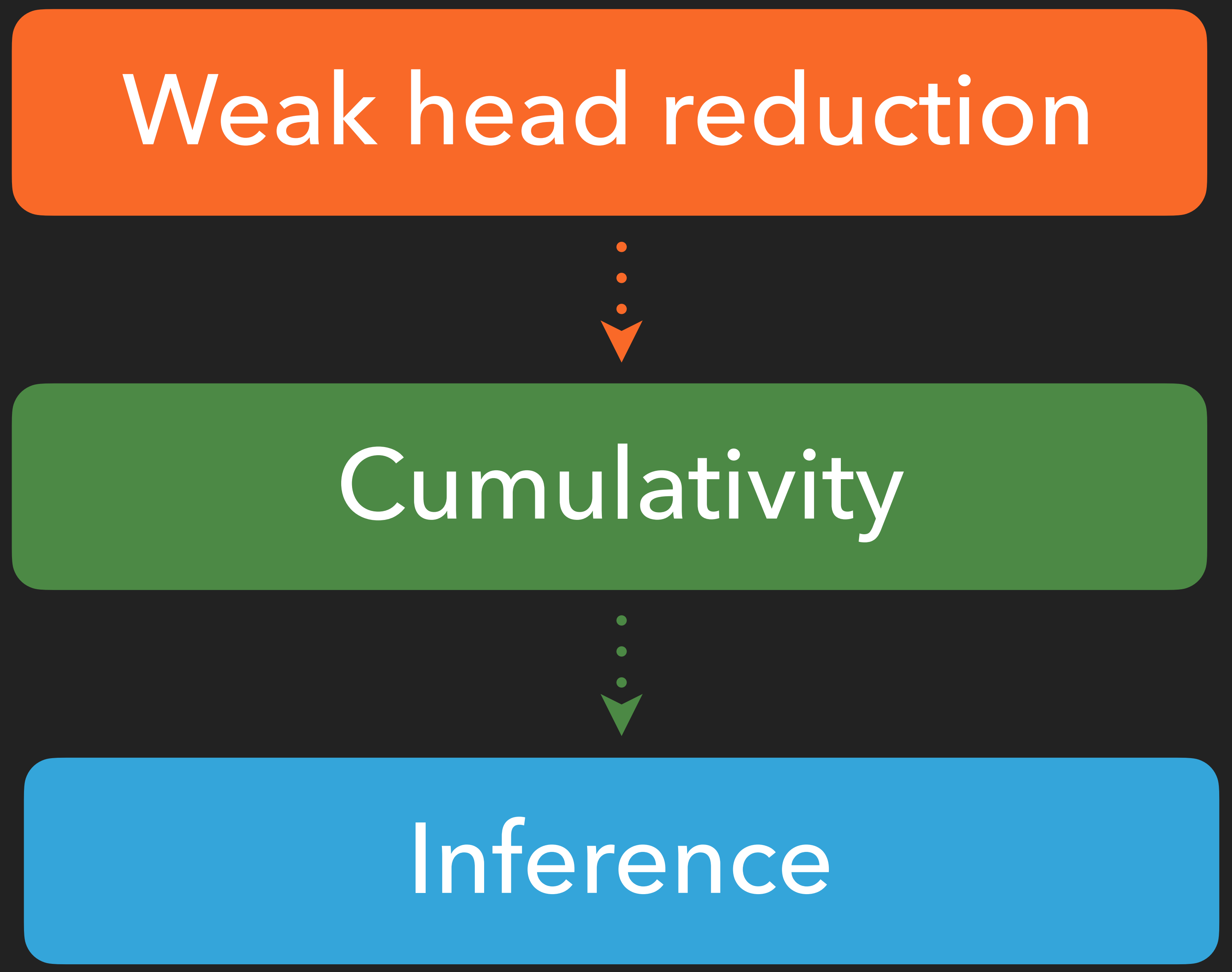
Cumulativity

Inference

Type Checking

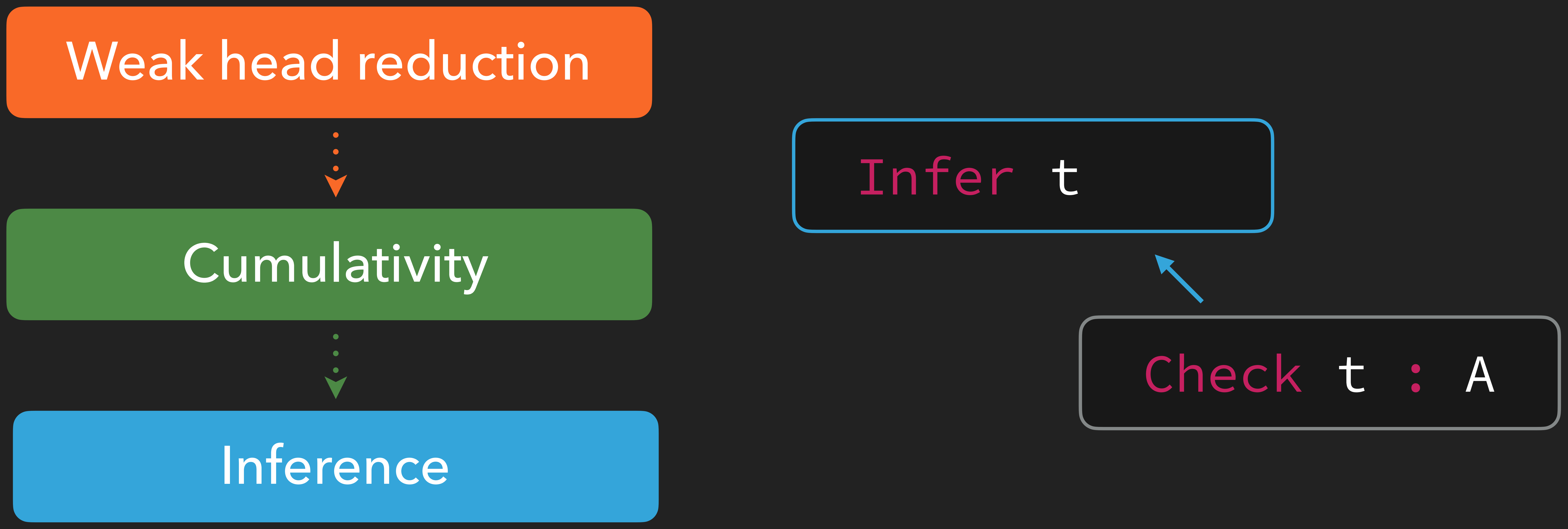


Type Checking




```
Check t : A
```

Type Checking



Type Checking

Weak head reduction

Cumulativity

Inference

`Infer t : B`

`Check t : A`

Type Checking

Weak head reduction

Cumulativity

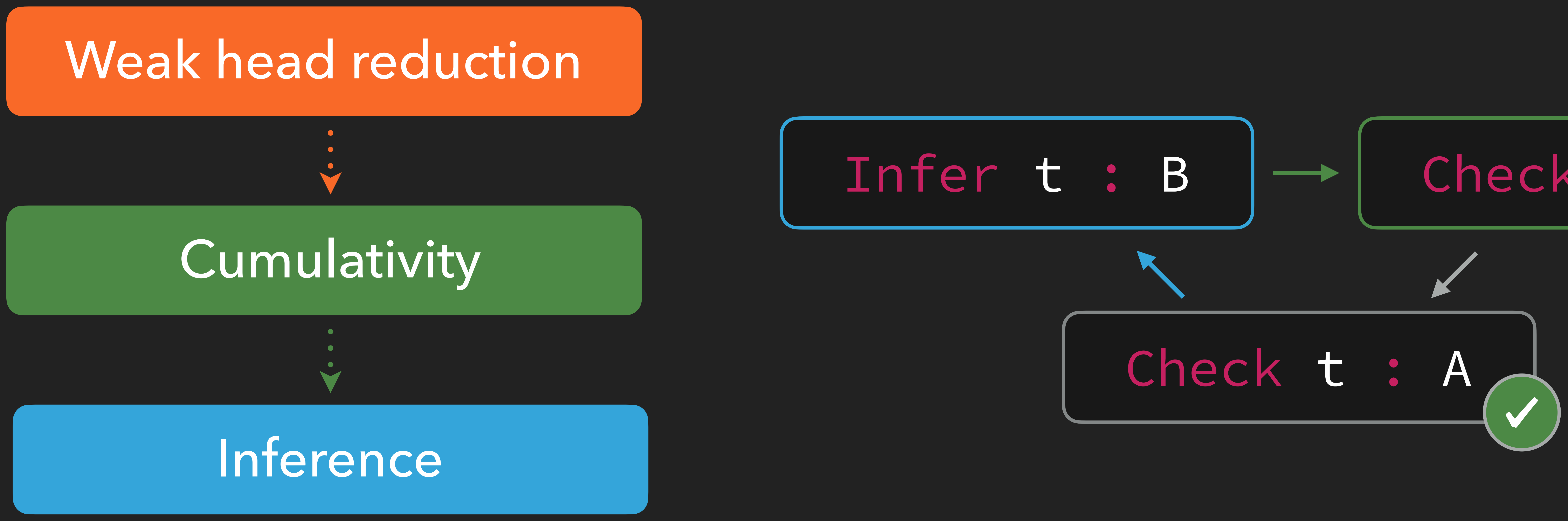
Inference

`Infer t : B`

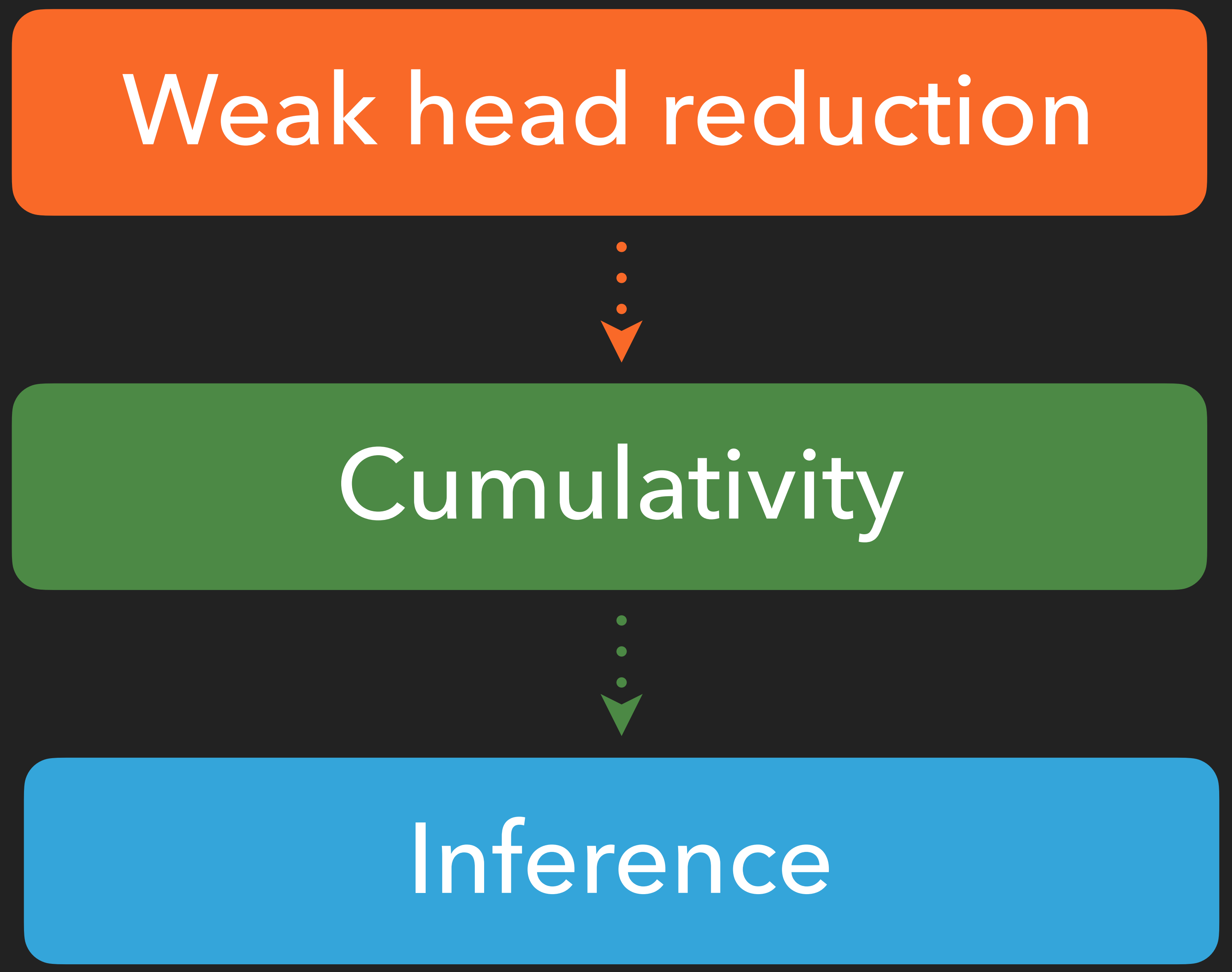
`Check B ≤ A`

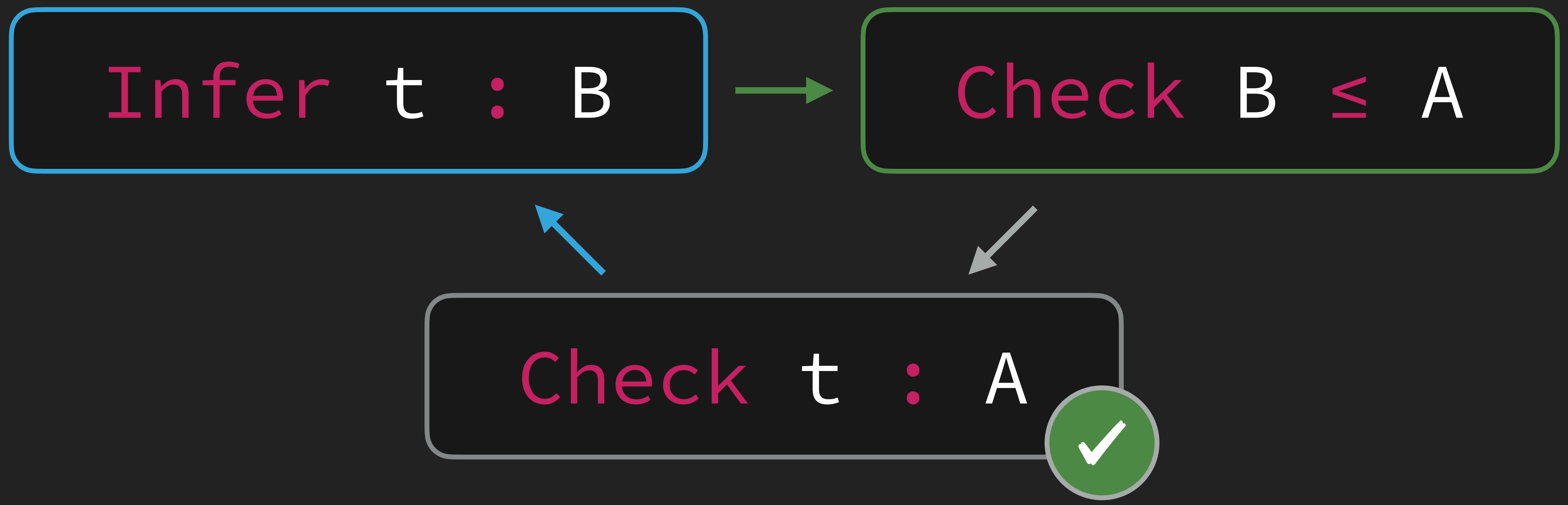
`Check t : A`

Type Checking



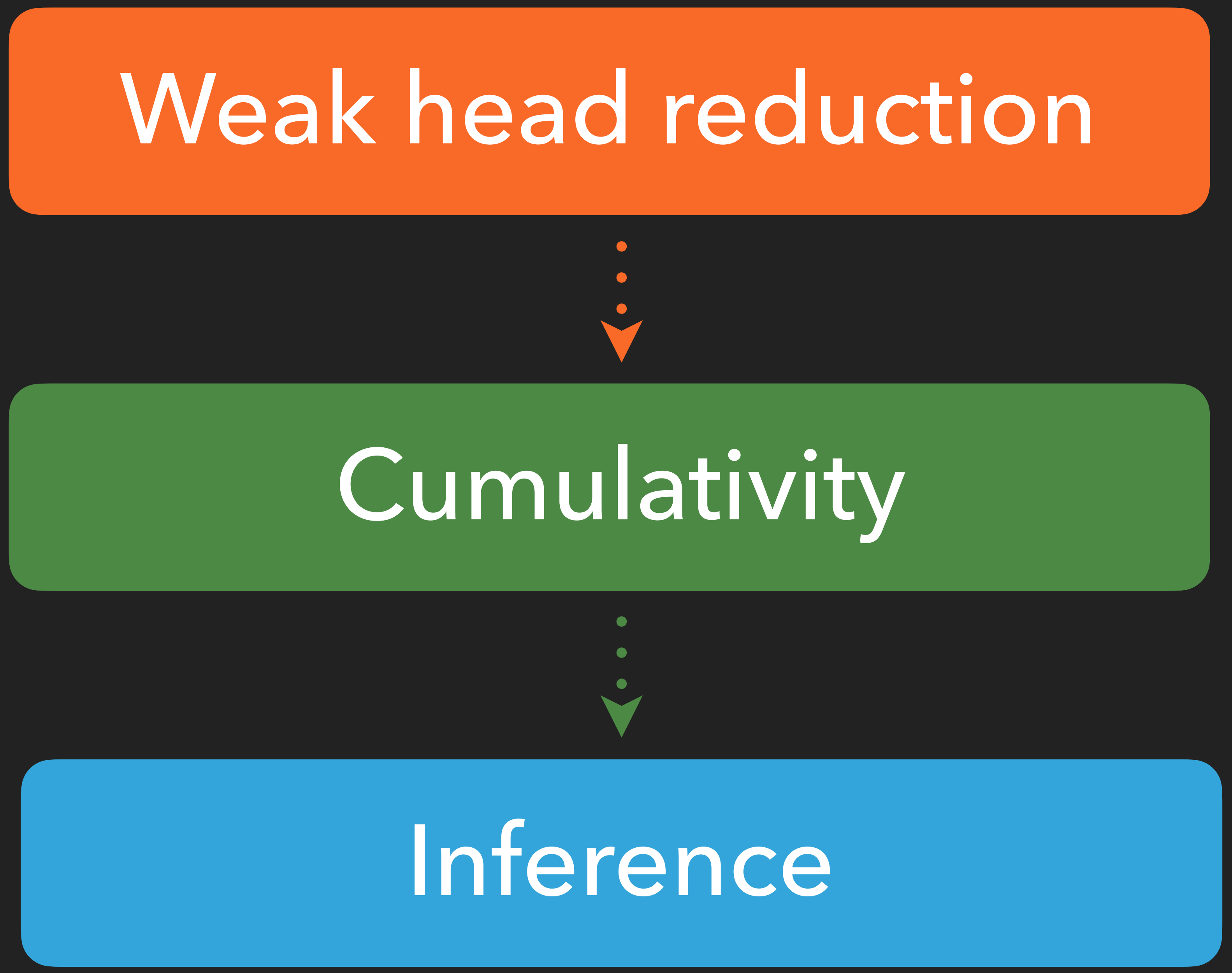
Type Checking

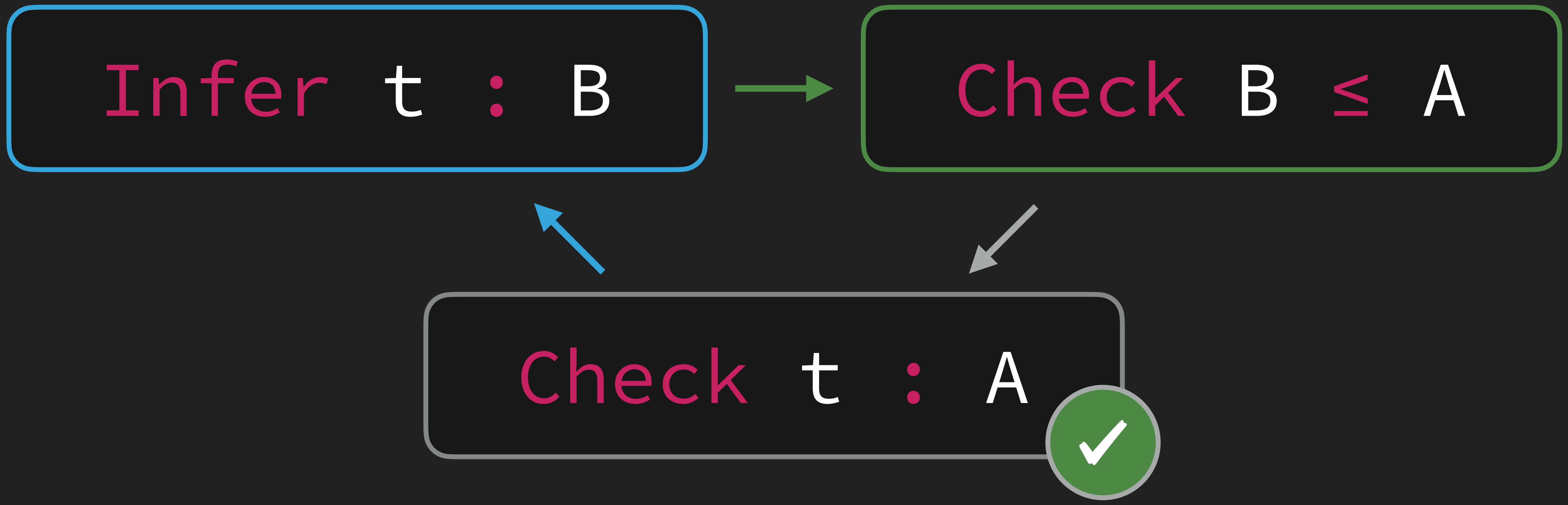





```
MetaCoq Check foo.
```

Type Checking






```
MetaCoq Check foo.
```

WIP Bidirectional type checking for completeness

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

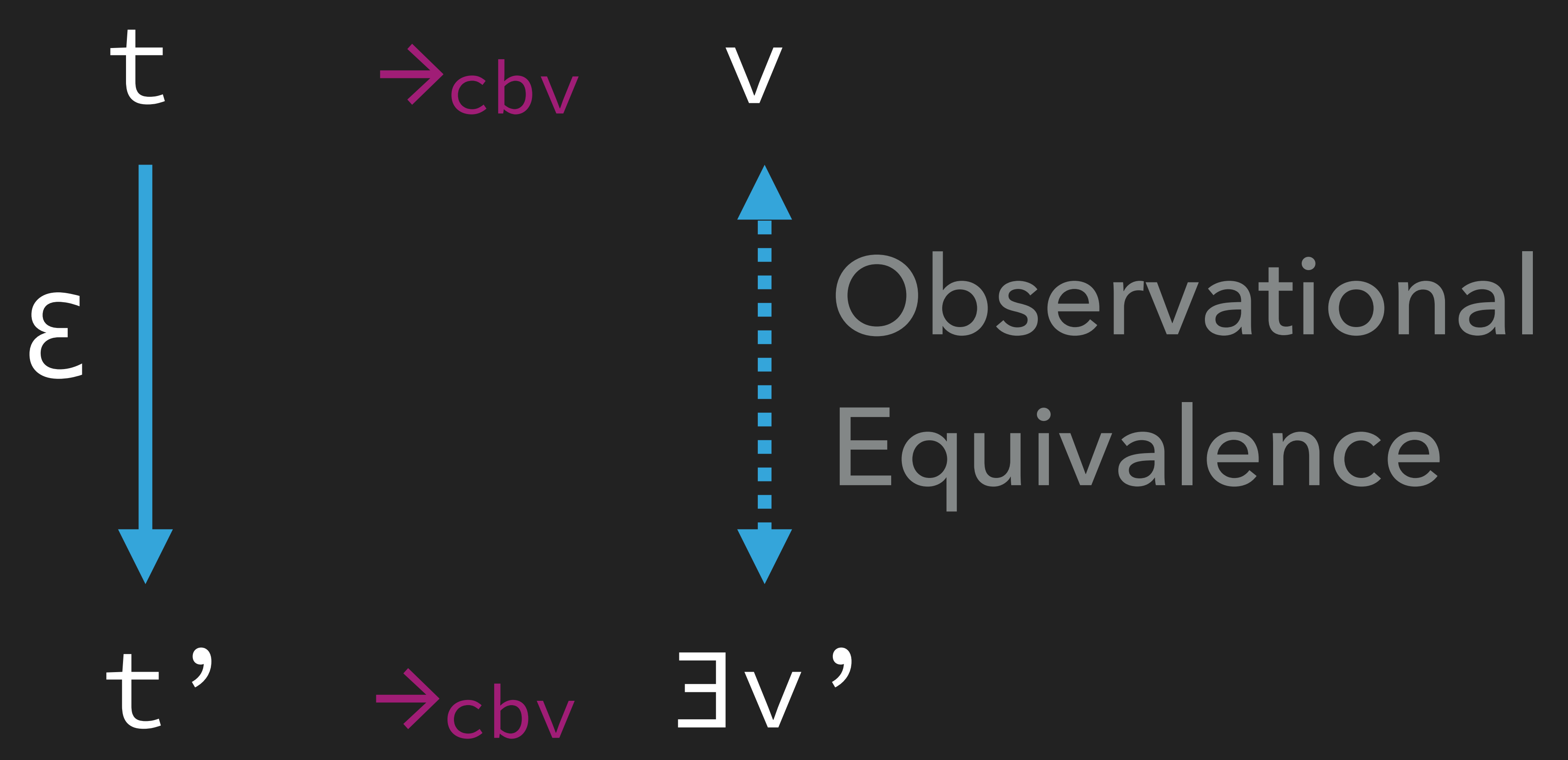
Singleton elimination principle

Erase propositional content used in computational content:

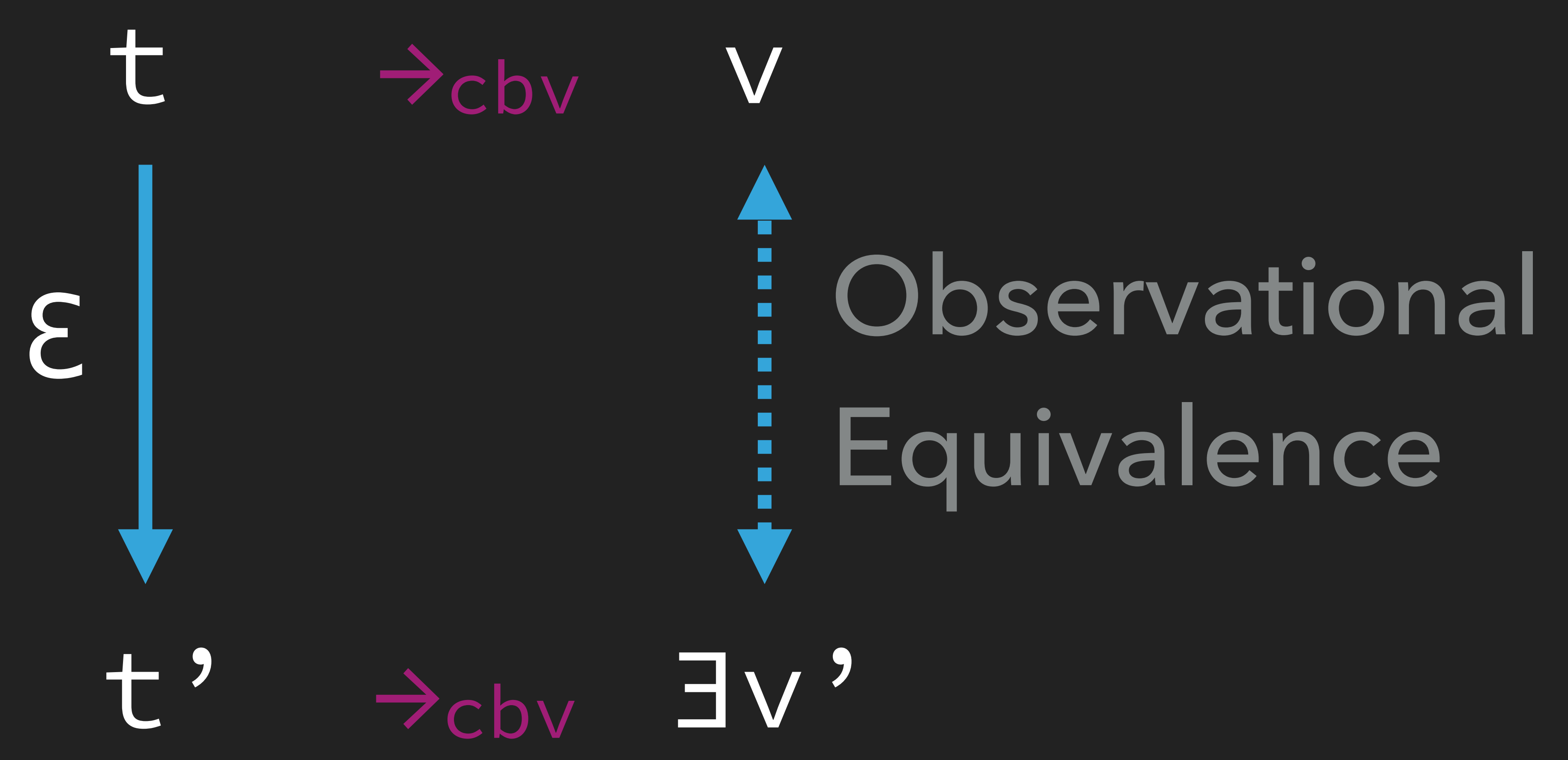
```
Ɛ (match p in eq _ y with eq_refl ⇒ b end) = Ɛ (b)
```

```
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



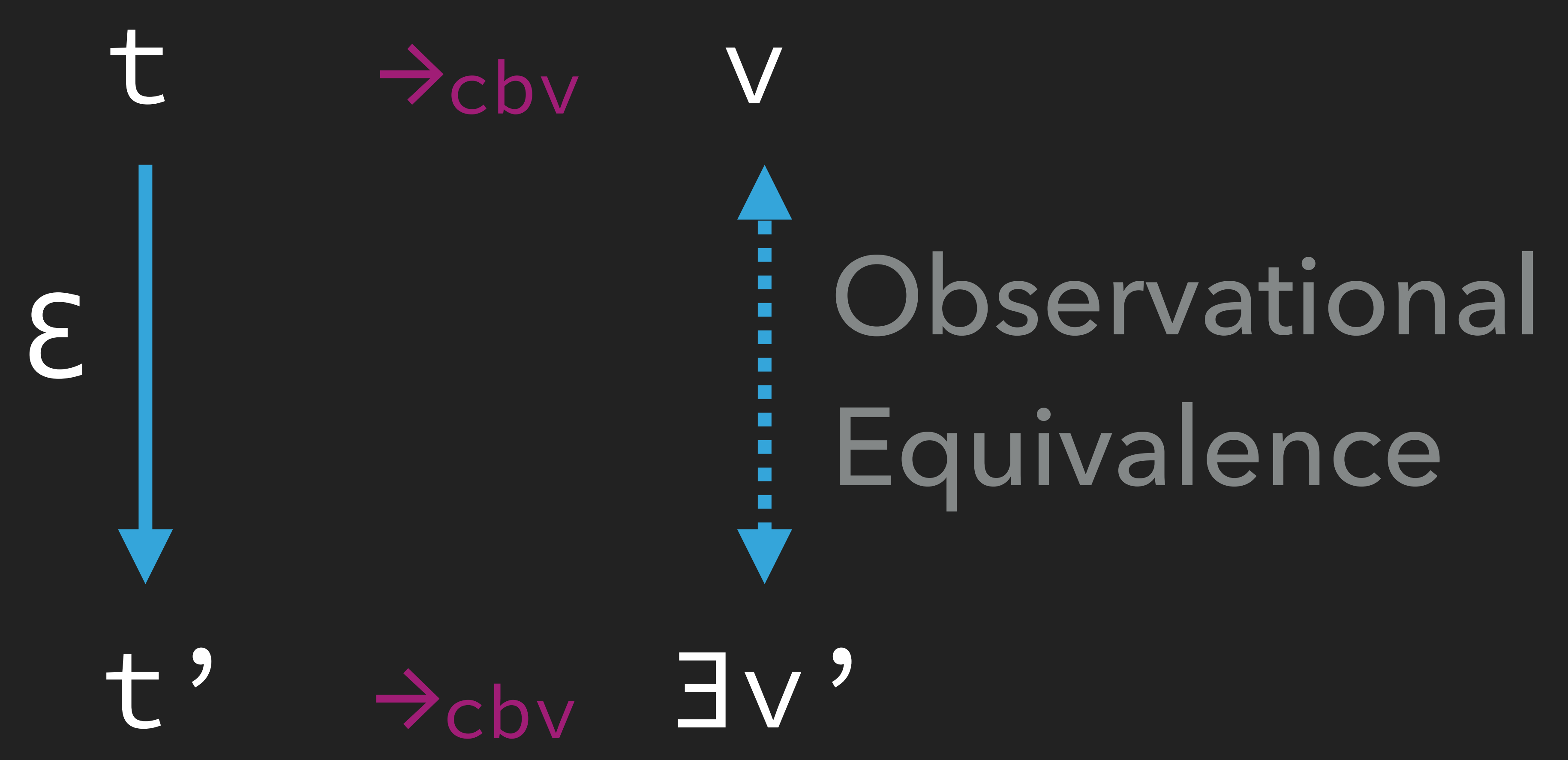
Erasure Correctness



With Canonicity and SN:

$\vdash t : \text{nat}$

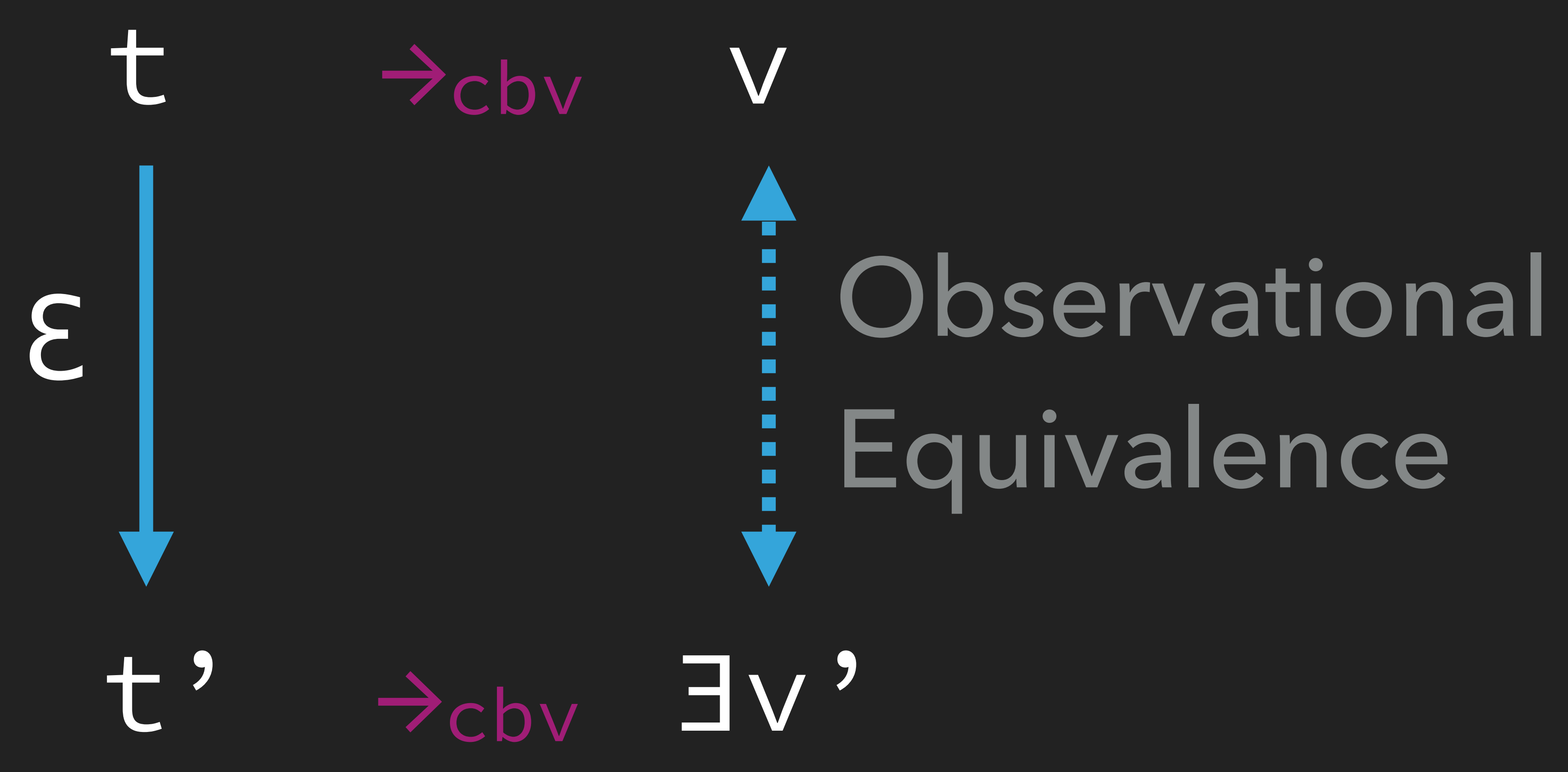
Erasure Correctness



With Canonicity and SN:

```
   ⊢ t : nat
=> ⊢ t → n : nat    (n ∈ ℕ)
```

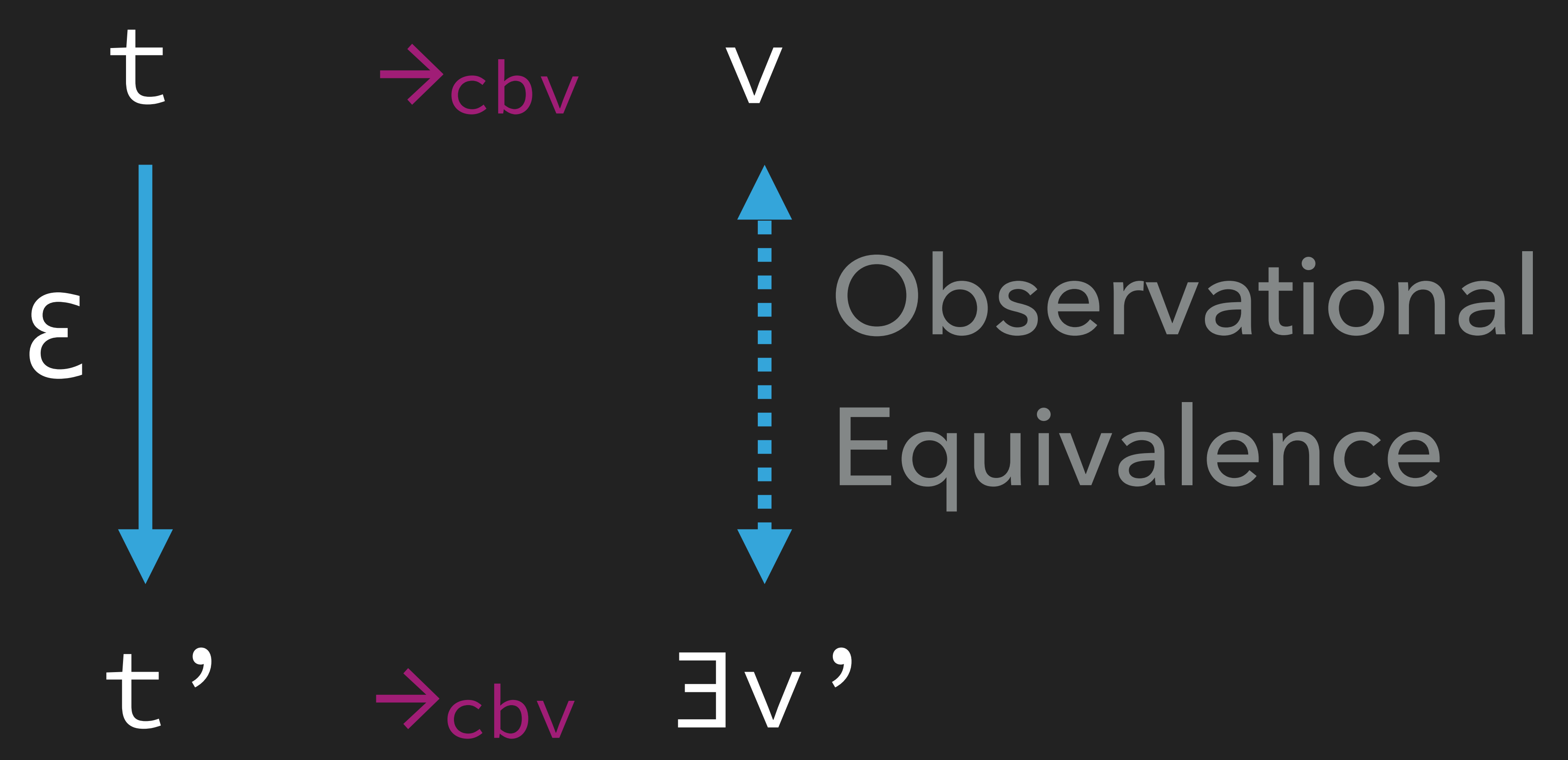
Erasure Correctness



With Canonicity and SN:

```
   ⊢ t : nat
=> ⊢ t → n : nat    (n ∈ ℕ)
=> t →cbv n : nat
```

Erasure Correctness



With Canonicity and SN:

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

Erasure Correctness

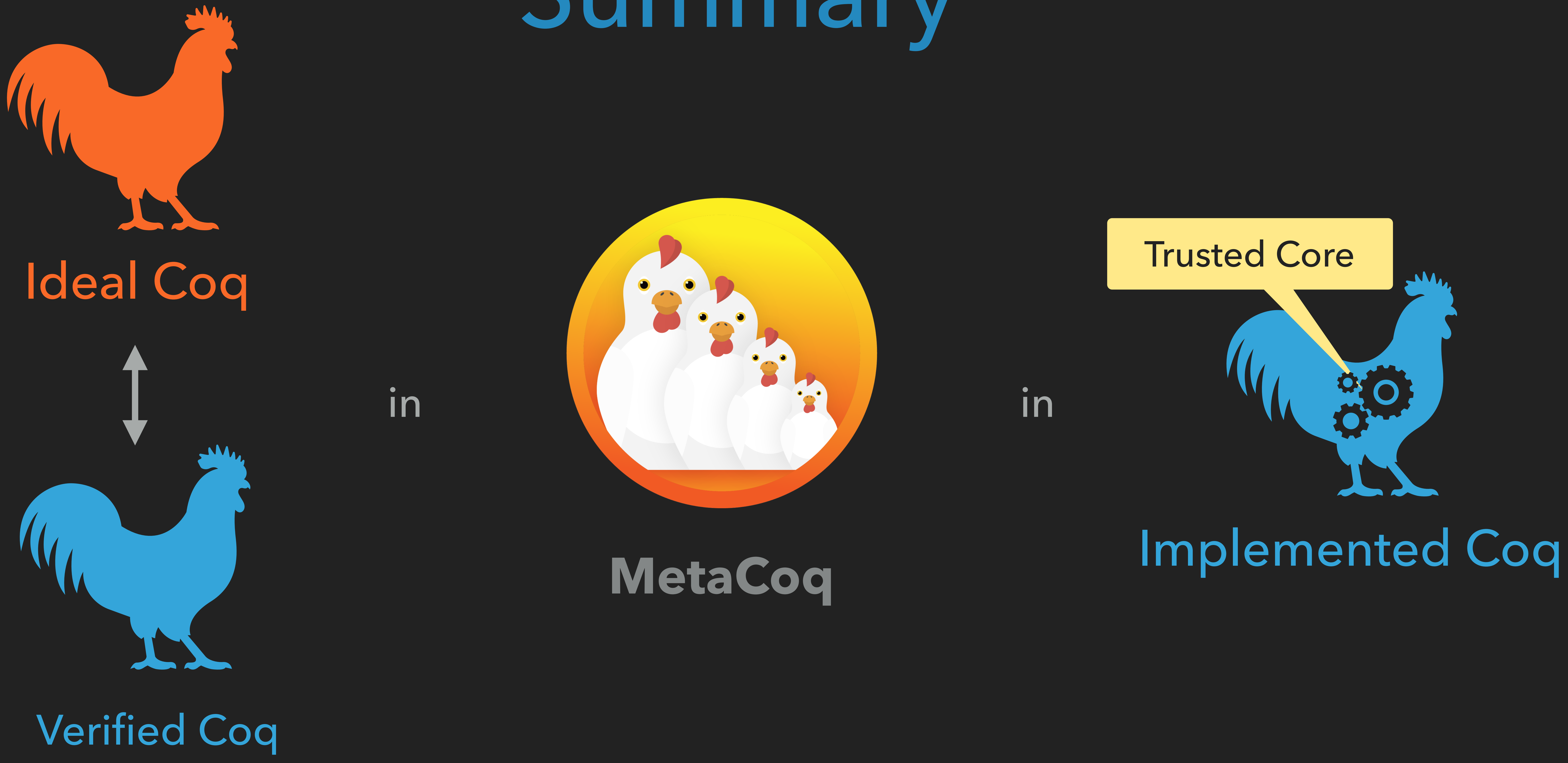
First define a non-deterministic erasure relation, then define:

```
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:

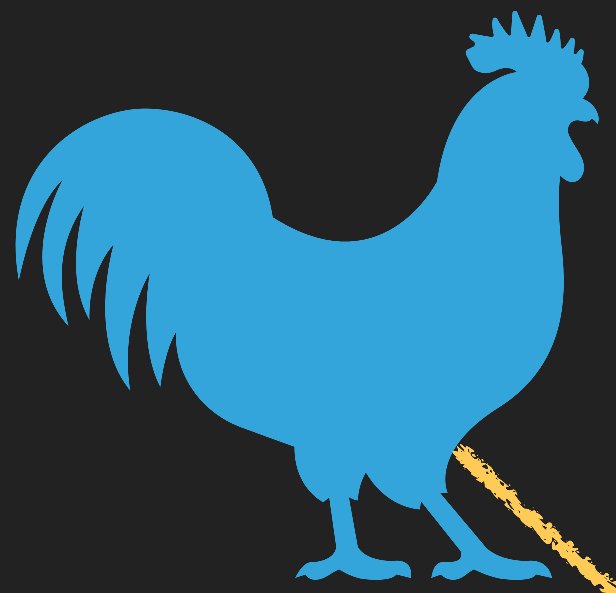
- Remove trivial cases on singleton inductive types in `Prop`
- Compute the dependencies of the erased term to erase only the computationally relevant subset of the global environment. I.e. remove unnecessary proofs the original term depended on.

Summary



Ideal Coq
Verified Coq
in
MetaCoq
in
Trusted Core
Implemented Coq

Summary



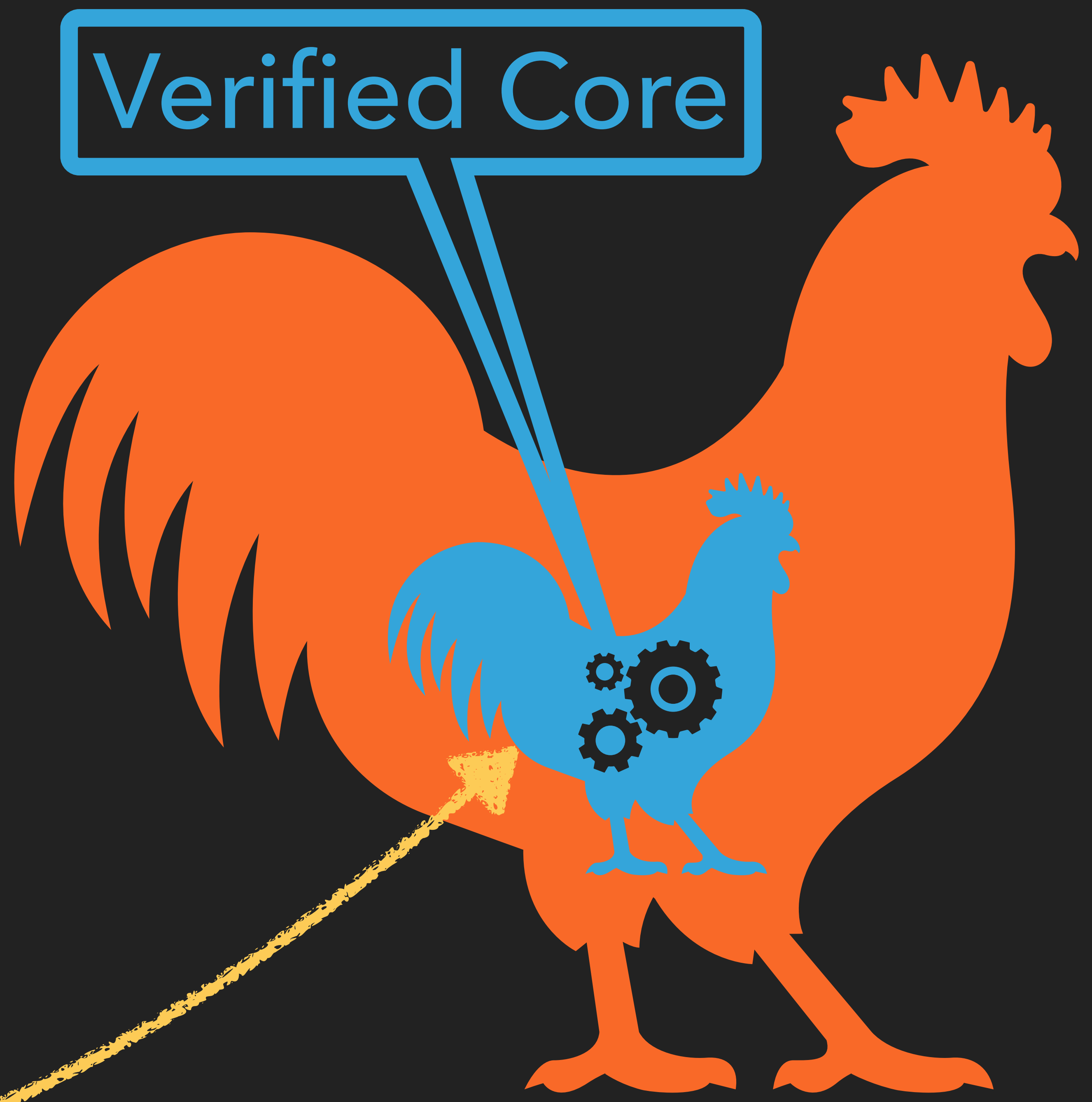
Verified Coq

```
MetaCoq Check vrev.
```

in

MetaCoq

in



Implemented Coq

=

Ideal Coq

Spec: 30kLoC

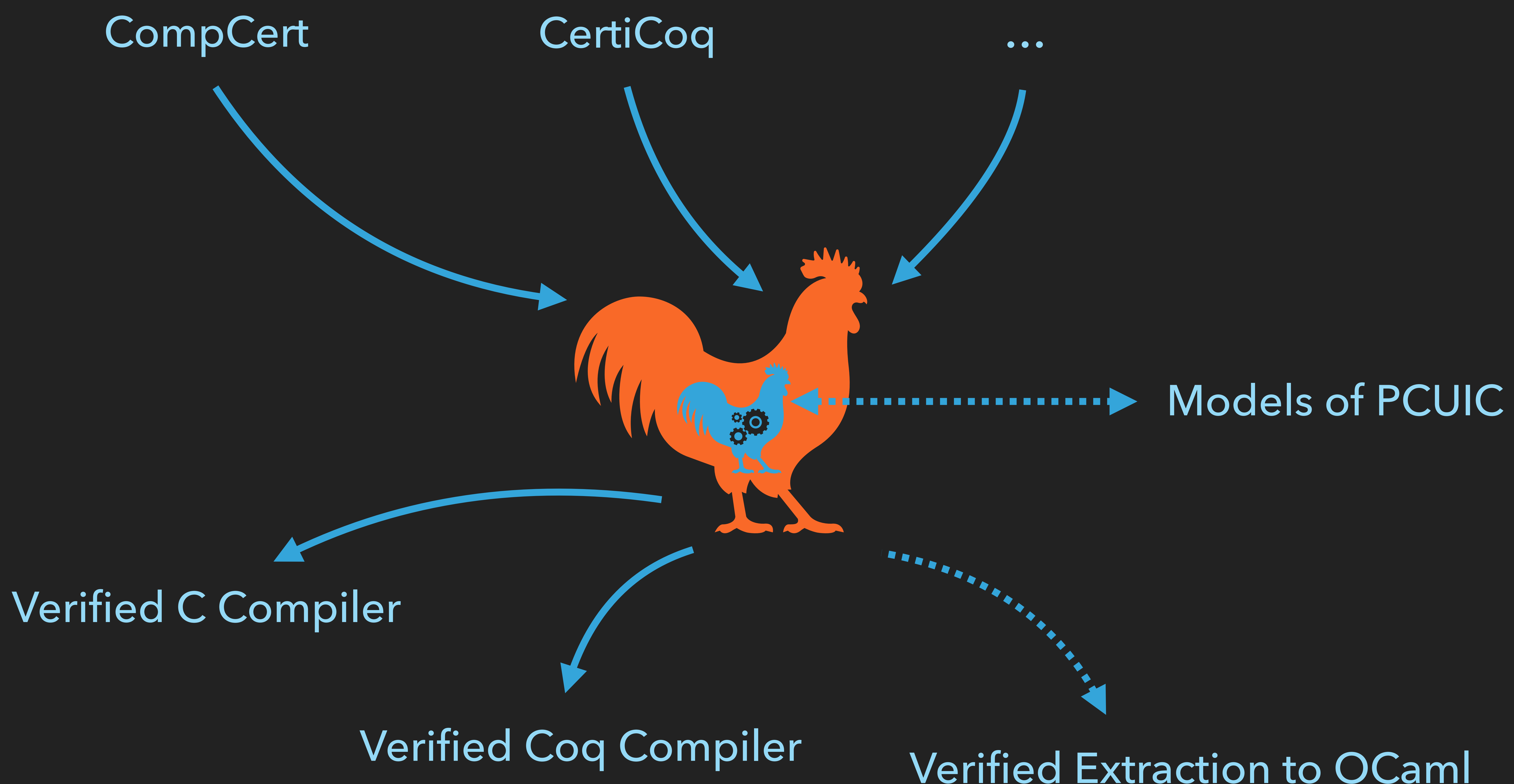
Proofs: 60kLoC

Comments: 10kLoC

Verified ℰ

```
MetaCoq Erase vrev.
```

Perspectives



A little success story

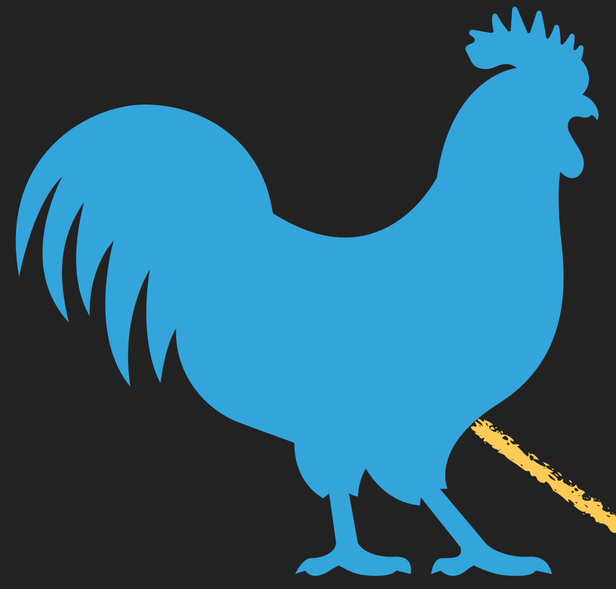
Spec/Proof/Program co-design for the new `match` representation in Coq (PR #13563 by P.M. Pédrot, CEP #34 by H. Herbelin).

- MetaCoq => typechecking of case on cumulative inductive types is incomplete
- Failure of subject reduction in Coq.
- "Quick" fix requires strengthening which in turn is not provable without subject reduction, leading to a messy meta-theory. Also incompatible with eta-conversion.
- The new representation solves all these issues **and** reflects the high-level user syntax more faithfully. It's win/win/win!

Ongoing and future work

- Integration of **rewrite rules** (CEP #50)
- Interoperability of erased code with OCaml (Nomadic Labs **CoqExtra** project, Pierre Giraud's PhD thesis)
- Full meta-theory for the **SProp** sort and irrelevance checking
- **Eta-reduction** and contravariant subtyping (CEP #47)
- Integration of a **sort-polymorphism** system, generalising universe polymorphism to deal more uniformly with impredicative sorts and alternative hierarchies (exceptional type theory, setoid type theory, erasable sets…) (Kenji Maillard).

Conclusion

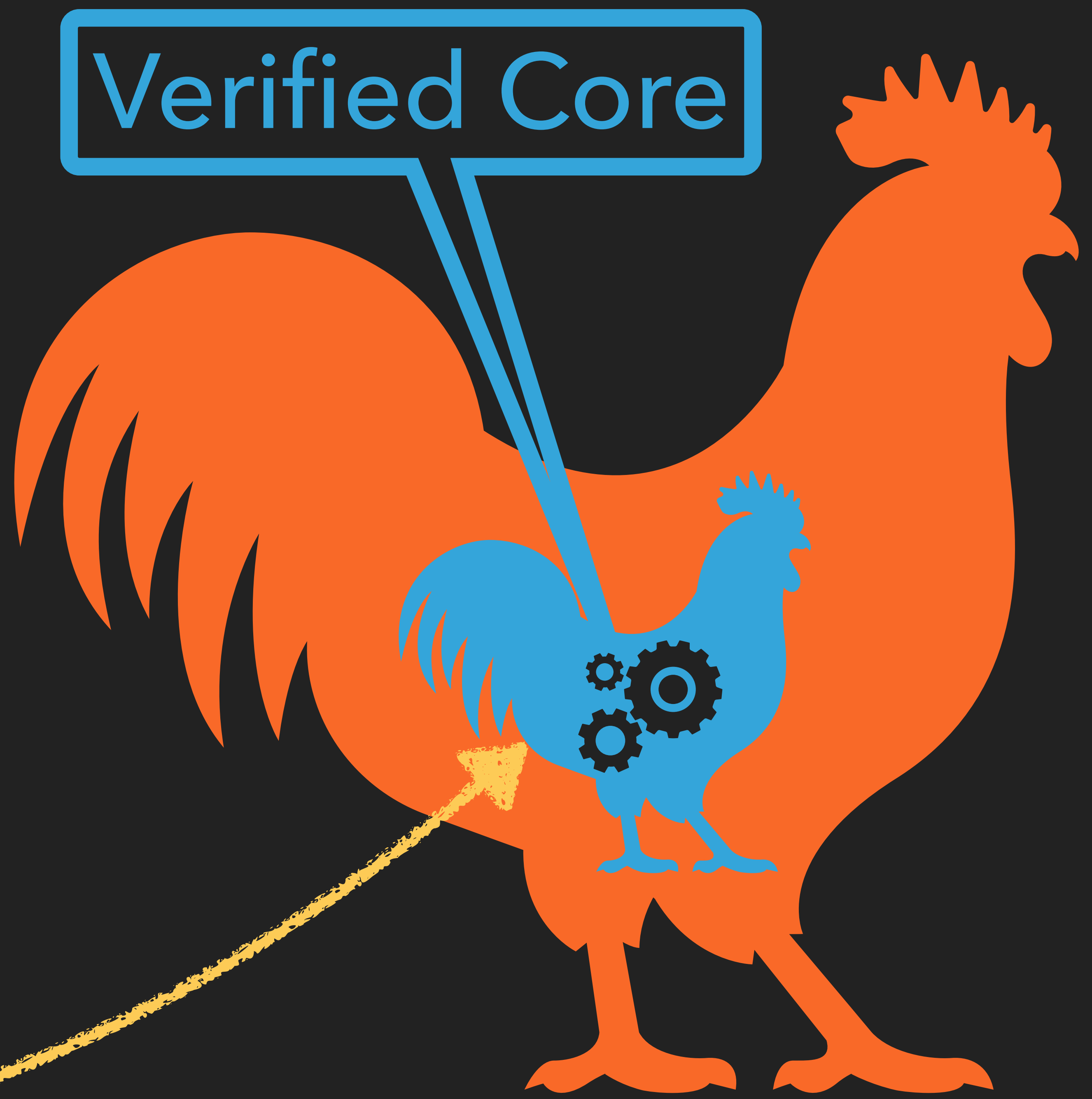


Verified Coq

in

MetaCoq

in



Implemented Coq

=

Ideal Coq

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

Verified ε

https://metacoq.github.io

Coq in MetaCoq

« *Cot Cot Codet* ». French, Interjection.

1. Cackle (the cry of a hen, especially one that has laid an egg).

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)