POPL 2020

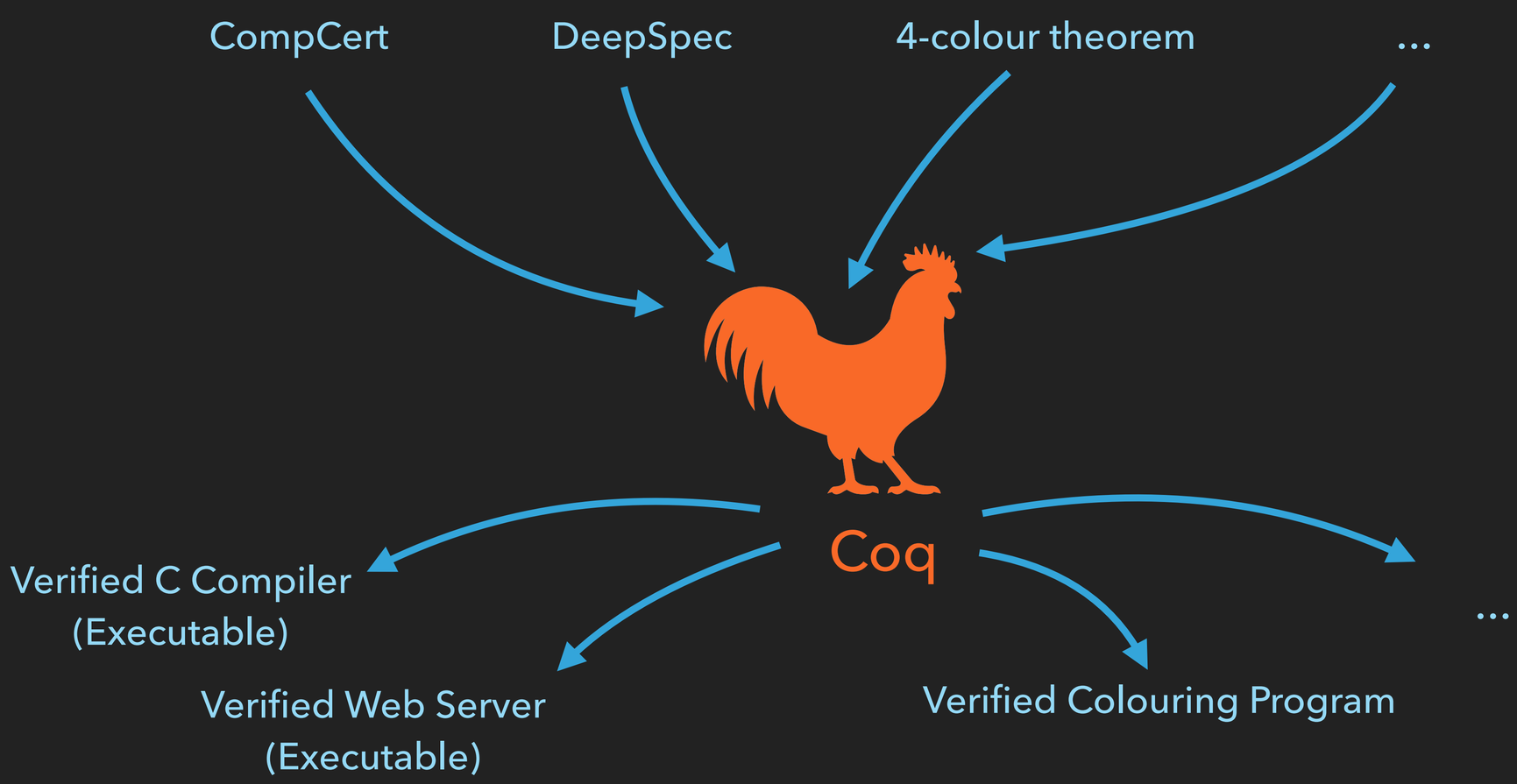
Coq Coq Correct!

Verification of Type Checking and Erasure for Coq, in Coq

Matthieu **Sozeau** Simon **Boulier** Yannick **Forster**

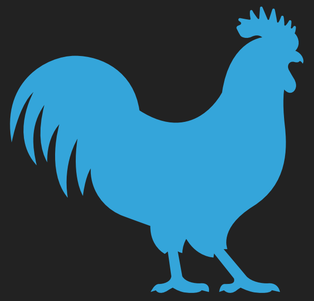
Nicolas **Tabareau** Théo **Winterhalter**

Motivation


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

What do you trust?

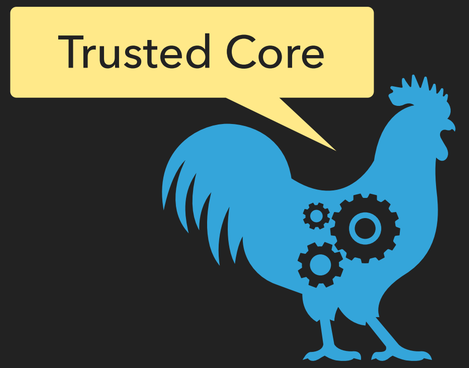
Ideal Coq



Implemented Coq

What do you trust?

Ideal Coq

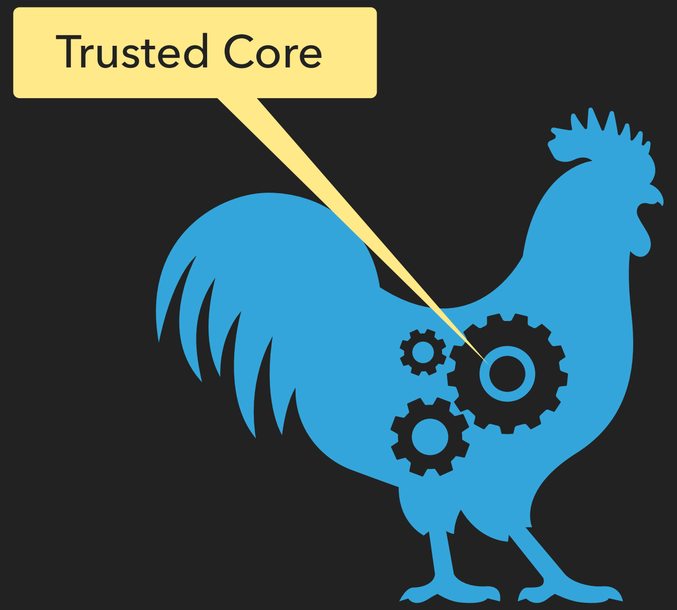


Implemented Coq

What do you trust?

Dependent Type Checker (18kLoC, 30+ years)

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

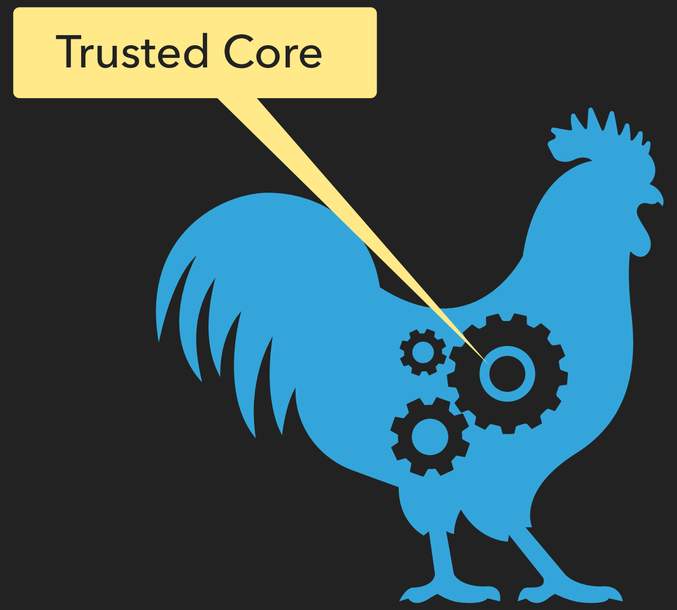


Implemented Coq

What do you trust?

Dependent Type Checker (18kLoC, 30+ years)

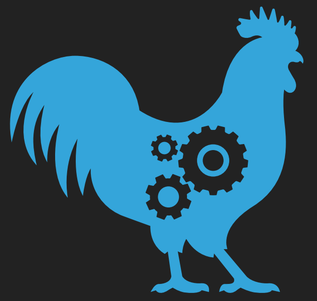
- Inductive Families w/ Termination Checker
- Universe Cumulativity and Polymorphism
- ML-style Module System
- KAM, VM and Native Conversion Checkers
- OCaml's Compiler and Runtime



Implemented Coq

The Reality

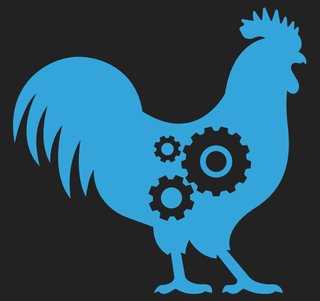
Ideal Coq



Implemented Coq

The Reality

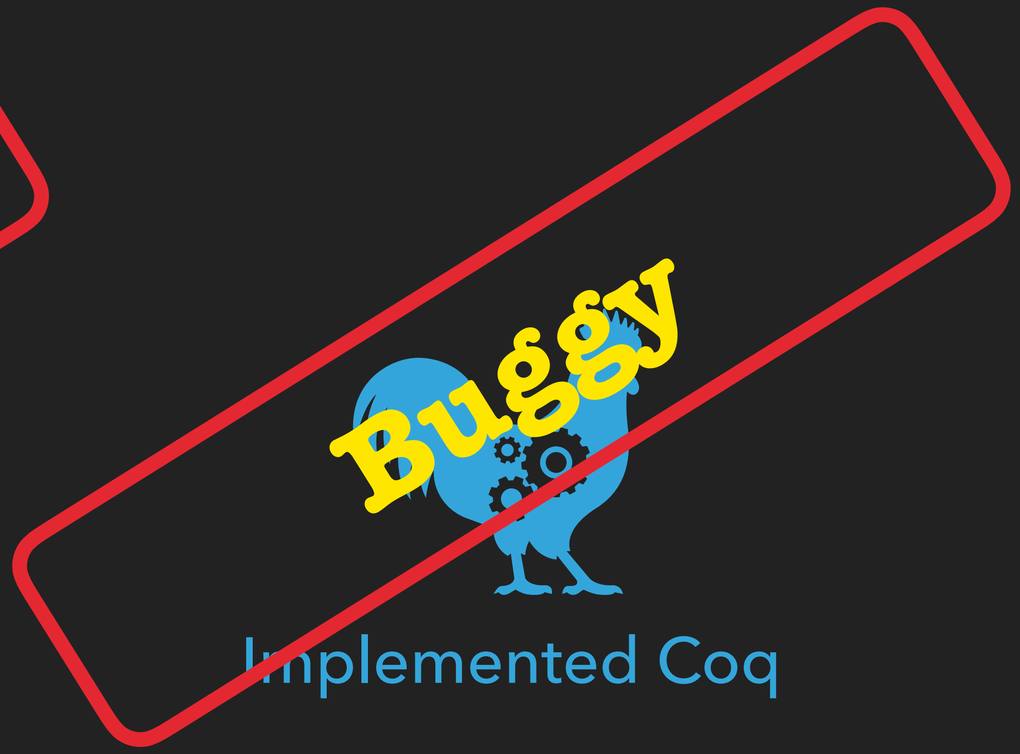
Ideal Coq



Implemented Coq

The Reality

Ideal Coq



Implemented Coq

The Reality

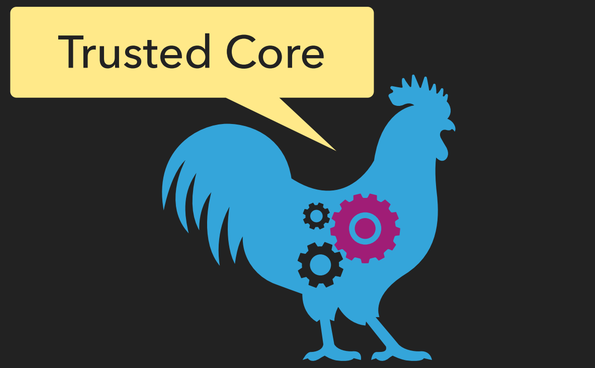
354 lines (314 sloc) | 16.7 KB

```
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
```

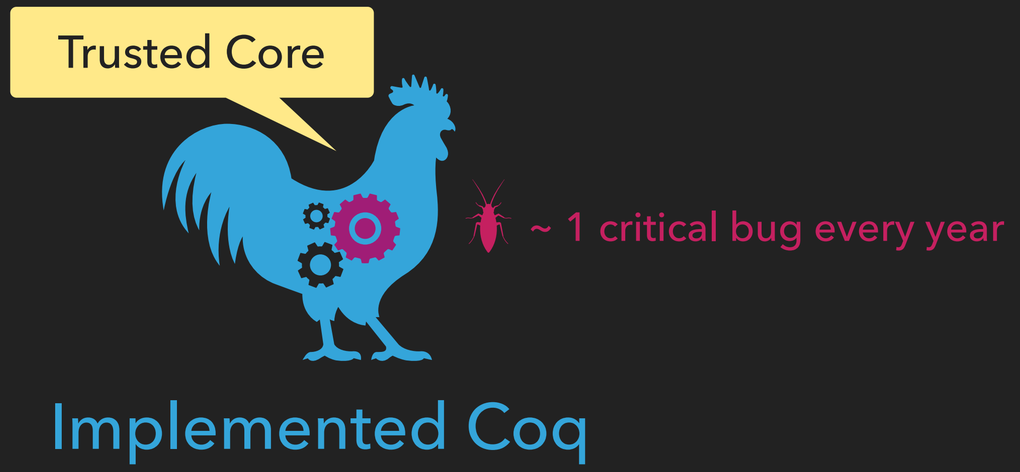


Implemented Coq

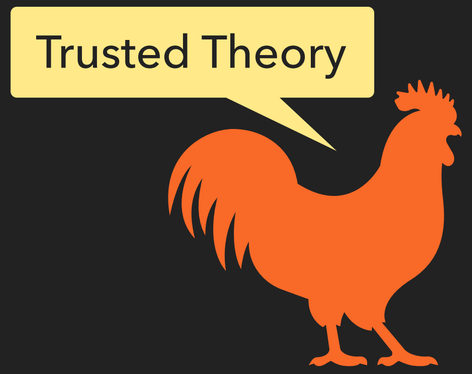
The Reality

354 lines (314 sloc) | 16.7 KB

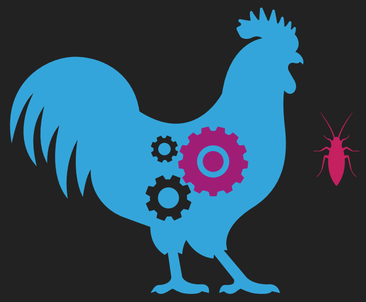
```
1  Preliminary compilation of critical bugs in stable releases of Coq
2  ================================================================
3    WORK IN PROGRESS WITH SEVERAL OPEN QUESTIONS
4
5
6  To add: #7723 (vm_compute universe polymorphism), #7695 (modules an
7
8  Typing constructions
9
10   component: "match"
11   summary: substitution missing in the body of a let
12   introduced: ?
13   impacted released versions: V8.3-V8.3pl2, V8.4-V8.4pl4
14   impacted development branches: none
15   impacted coqchk versions: ?
16   fixed in: master/trunk/v8.5 (e583a79b5, 22 Nov 2015, Herbelin), v
17   found by: Herbelin
```



Our Goal: Improving Trust



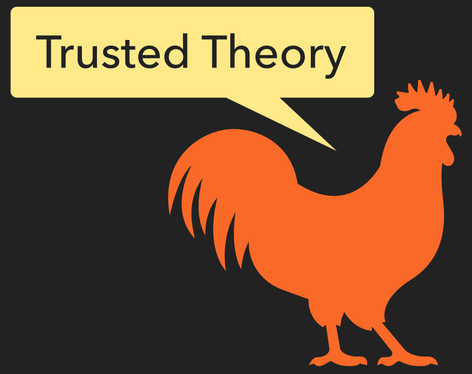
Ideal Coq



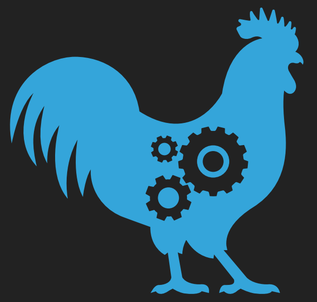
~ 1 critical bug every year

Implemented Coq

Our Goal: Improving Trust

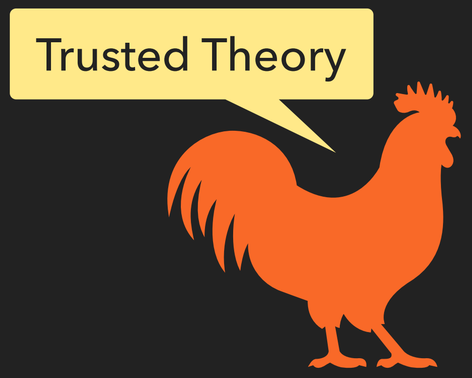


Ideal Coq

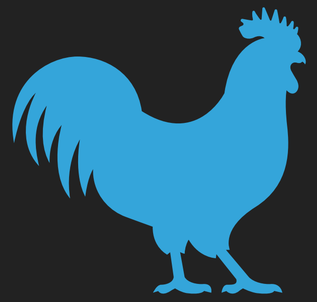


Implemented Coq

Coq in MetaCoq



Coq Spec: PCUIC



Verified Coq

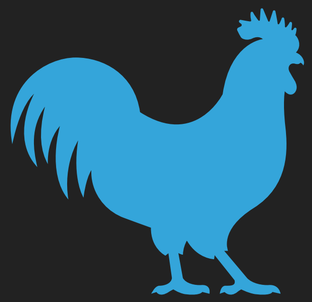
in

MetaCoq

Formalization of Coq in Coq

JAR'20

in



Implemented Coq

What we have...

```
fix vrev {A : Type@{i}} {n m : nat} (v : vec 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.
```

What we have...

```
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.
```

```
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 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.
```

What we have...

```
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.
```

What we have...

```
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.
```

What we have...

```
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.
```

What we have...

```
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.
```

What we have...

```
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.
```

...and what we don't

```
(fun x ⇒ f x) ≡ f   (x ∉ f)
```

η-conversion

```
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

```
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

```
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

```
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

```
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

Specification

Example: Reduction

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

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

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

Specification

Example: Reduction

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

Specification

Example: Reduction

DEFINITIONS IN CONTEXTS

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

GENERAL SUBSTITUTION

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

STRONG REDUCTION

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

Specification

Example: Reduction

DEFINITIONS IN CONTEXTS

GENERAL SUBSTITUTION

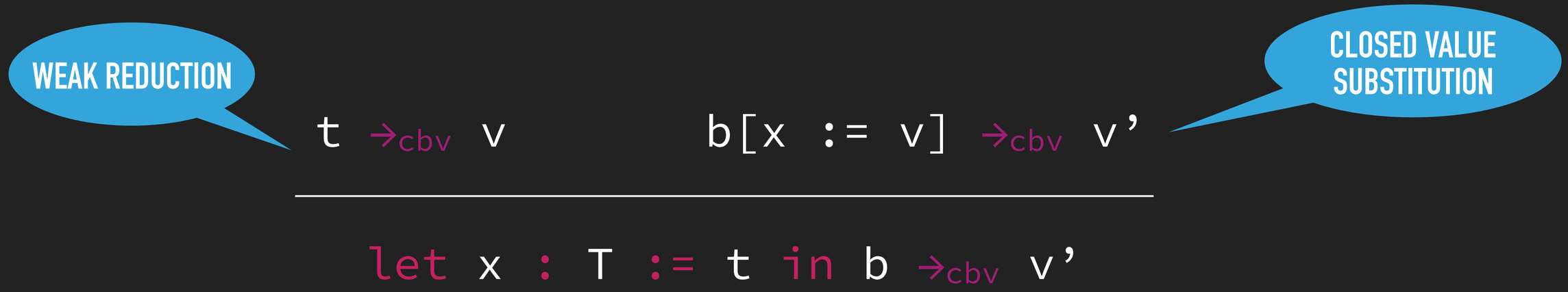
$$\frac{(x : T := t) \in \Gamma}{\Gamma \vdash x \to t} \qquad \frac{}{\Gamma \vdash \texttt{let}\ x : T := t\ \texttt{in}\ b \to b'[x := t]}$$

STRONG REDUCTION

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

Specification

Example: Call-by-Value Evaluation



$$\_ \to_{cbv} \_ \quad \subseteq \quad \varepsilon \vdash \_ \to \_$$

Meta-Theory

Proven Properties

Meta-Theory

Proven Properties

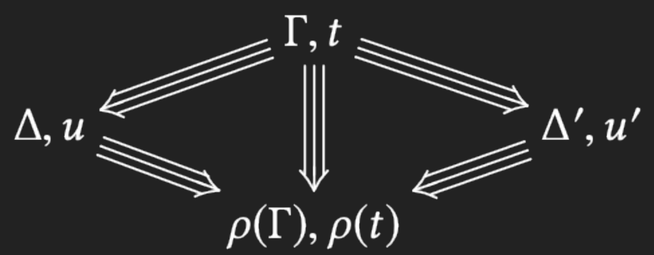
- ***Structural Properties***: substitution, local and global weakening, instantiation by universes

Meta-Theory

Proven Properties

- ***Structural Properties***: substitution, local and global weakening, instantiation by universes
- ***Confluence***

  à la Tait-Martin-Löf/Takahashi

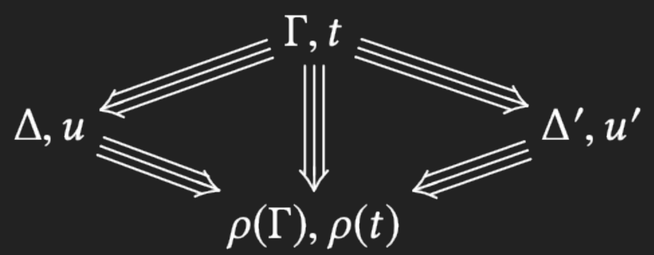


Meta-Theory

Proven Properties

- ***Structural Properties***: substitution, local and global weakening, instantiation by universes
- ***Confluence***

  à la Tait-Martin-Löf/Takahashi



- Inversion Principles

Trusted Theory Base

Assumptions

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch

- Strict Positivity & Guard Conditions
  Oracles: `check_fix : fixpoint -> bool`

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch
- Strict Positivity & Guard Conditions
  Oracles: `check_fix : fixpoint -> bool`
- *Subject Reduction & Principality*

Trusted Theory Base

Assumptions

- The specifications of typing, reduction and cumulativity
  ~ 500 LoC from scratch
- Strict Positivity & Guard Conditions
  Oracles: `check_fix : fixpoint -> bool`
- *Subject Reduction & Principality*
- Strong Normalization

Verifying Type-Checking

Reduction

Reduction

Input u

Reduction

Input u

Output v

Reduction

Input u

Output v

u → v

Reduction

Input `u : term`

Output `v` `u → v`

Reduction

Input `u : term`

Output `v : term`

`u → v`

Reduction

Input `u : term`

Output `v : term` `u → v : Prop`

Reduction

Input `u : term`

Output `v : term` `u → v : Prop`

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

Reduction

Reduction

Reduction

Reduction

Reduction

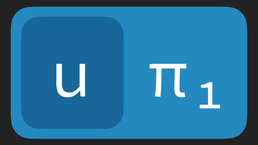
$t \longrightarrow \lambda x.x$

Reduction

$\lambda x.x \; u \; \pi_1$

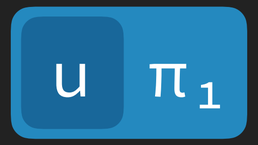
$t \longrightarrow \lambda x.x$

Reduction



$t \longrightarrow \lambda x.x$

Reduction



$$t\ u \longrightarrow (\lambda x.x)\ u \longrightarrow u$$

Reduction


```
match m with
| 0   ⇒ a
| S n ⇒ f n
end
```

Reduction


```
match m with
| 0     ⇒ a
| S n ⇒ f n
end
```

Reduction


```
match S p with
| 0     ⇒ a
| S n ⇒ f n
end
```

Reduction

Conversion

Specification

$$\frac{u \to w \qquad w \equiv v}{u \equiv v}$$

$$\frac{u \equiv_\alpha v}{u \equiv v}$$

$$\frac{u \equiv w \qquad v \to w}{u \equiv v}$$

Conversion

Algorithm

Conversion

Algorithm

Input `u : A`

Conversion

Algorithm

Input `u : A` `v : B`

Conversion

Algorithm

Input `u : A` `v : B`

Output `u ≡ v + error`

Conversion

Algorithm

Input `u : A` `v : B`

Output `u ≡ v + error`

```
isconv : ∀ (u v : term) → welltyped u → welltyped v → conv u v + error
```

Conversion

Algorithm

`u : A`

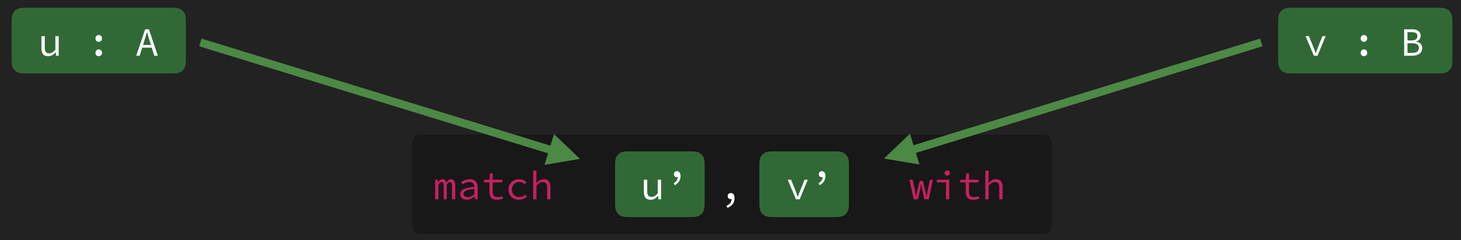
`v : B`

Conversion

Algorithm

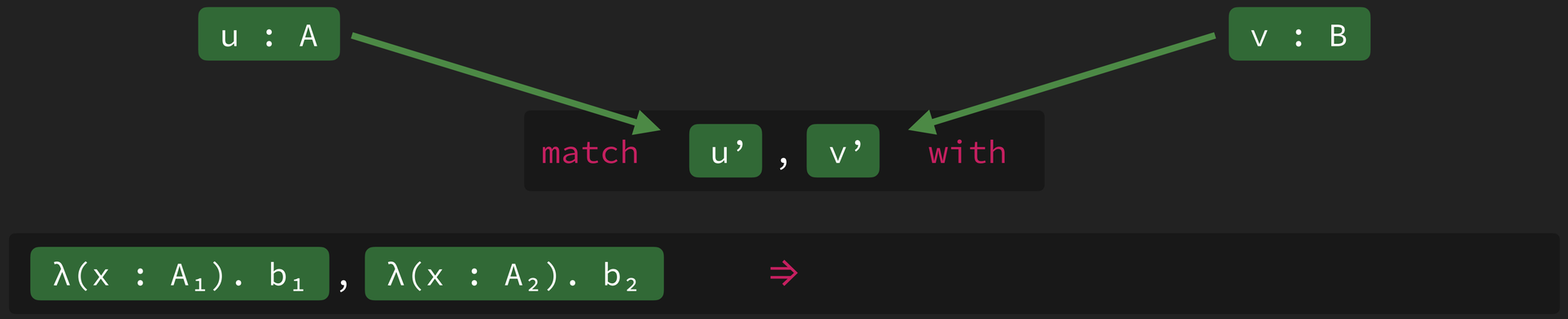
Conversion

Algorithm



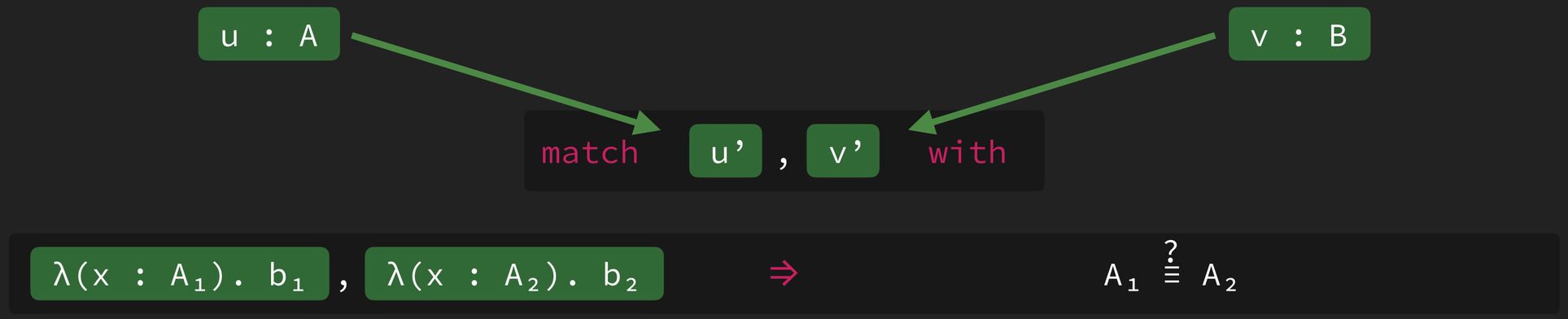
Conversion

Algorithm



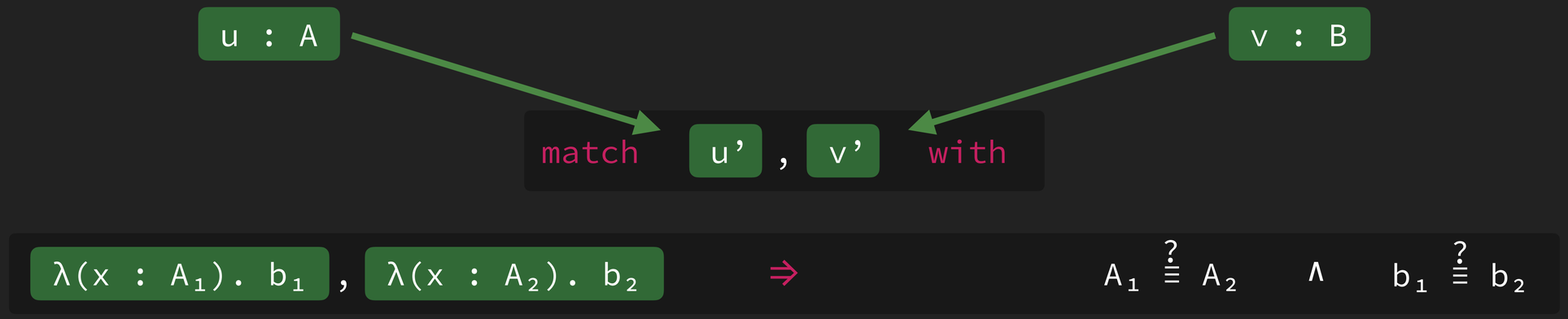
Conversion

Algorithm



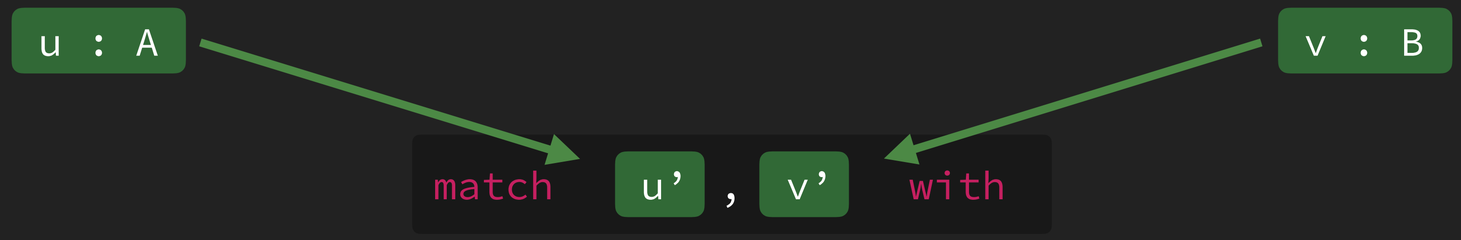
Conversion

Algorithm



Conversion

Algorithm

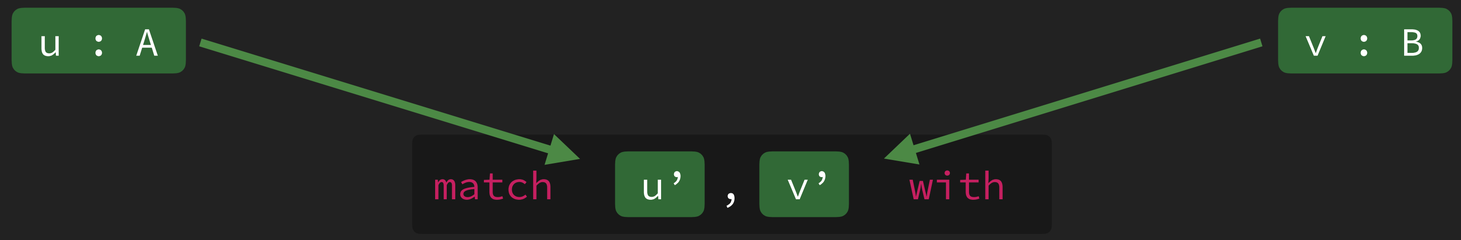


$\lambda(x : A_1).\ b_1$, $\lambda(x : A_2).\ b_2$ $\Rightarrow$ $A_1 \stackrel{?}{\equiv} A_2 \ \wedge \ b_1 \stackrel{?}{\equiv} b_2$

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

Conversion

Algorithm



$\lambda(x : A_1).\ b_1$, $\lambda(x : A_2).\ b_2$ $\Rightarrow$ $A_1 \stackrel{?}{\equiv} A_2 \ \wedge \ b_1 \stackrel{?}{\equiv} b_2$

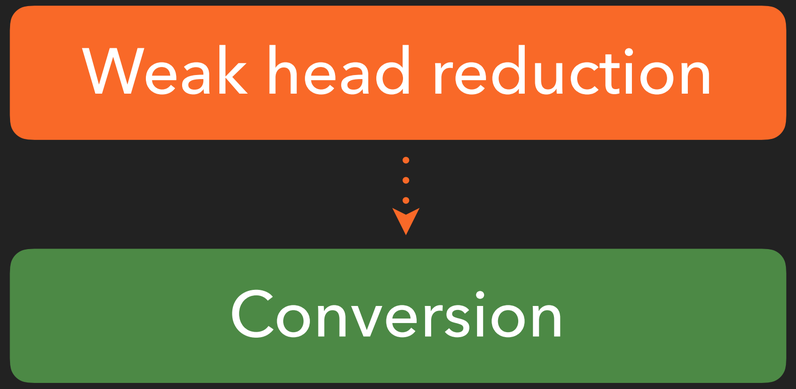
$\Pi(x : A_1).\ B_1$, $\Pi(x : A_2).\ B_2$ $\Rightarrow$ $A_1 \stackrel{?}{\equiv} A_2 \ \wedge \ B_1 \stackrel{?}{\equiv} B_2$

Type Checking

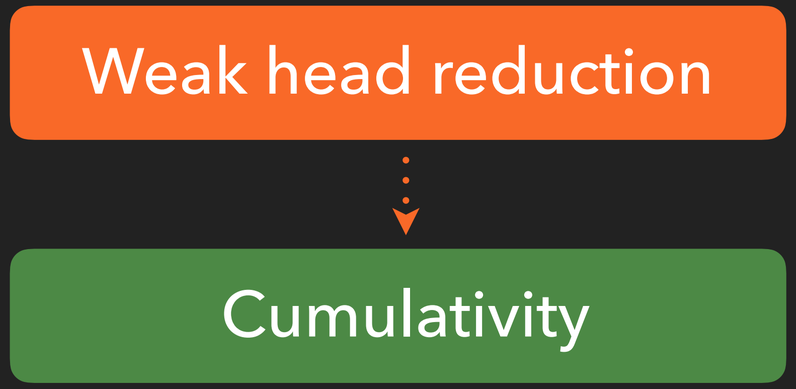
Type Checking

Weak head reduction

Type Checking

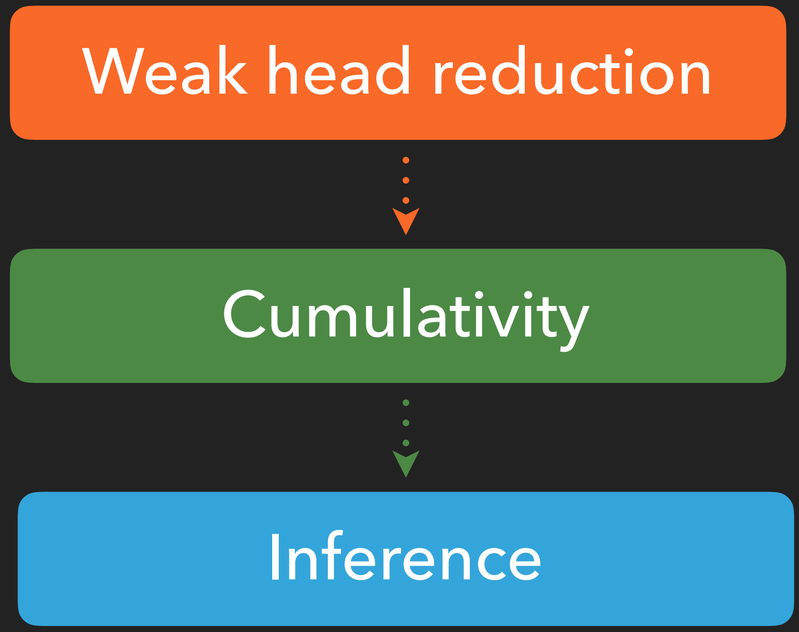


Type Checking



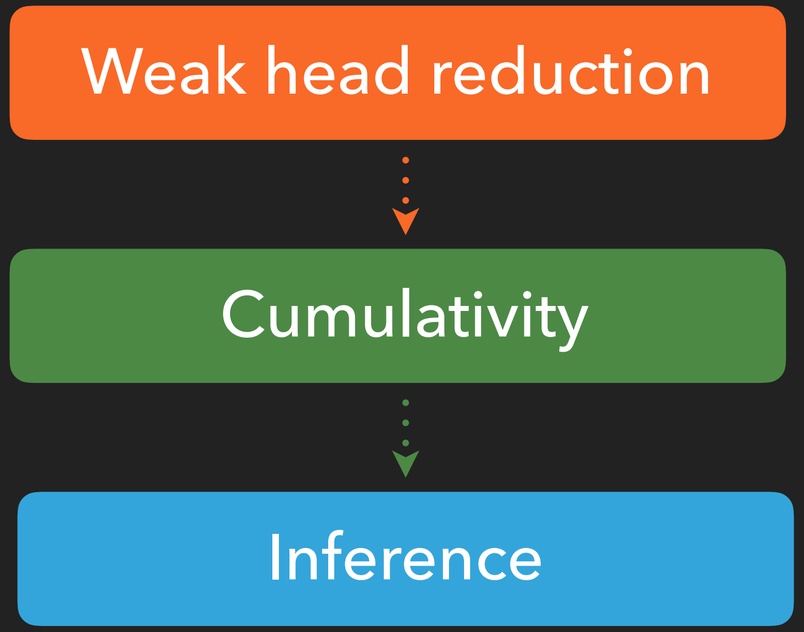
(Bellman-Ford)

Type Checking

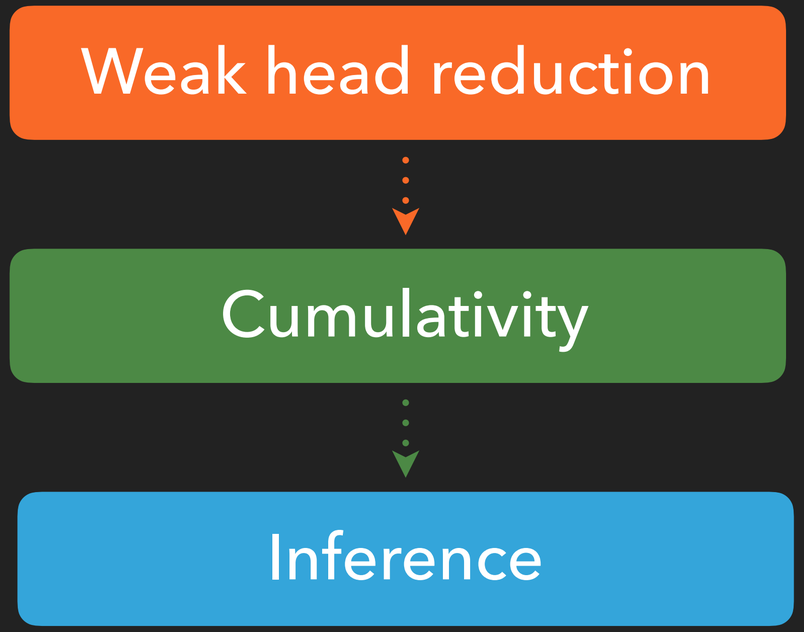


(Bellman-Ford)

Type Checking

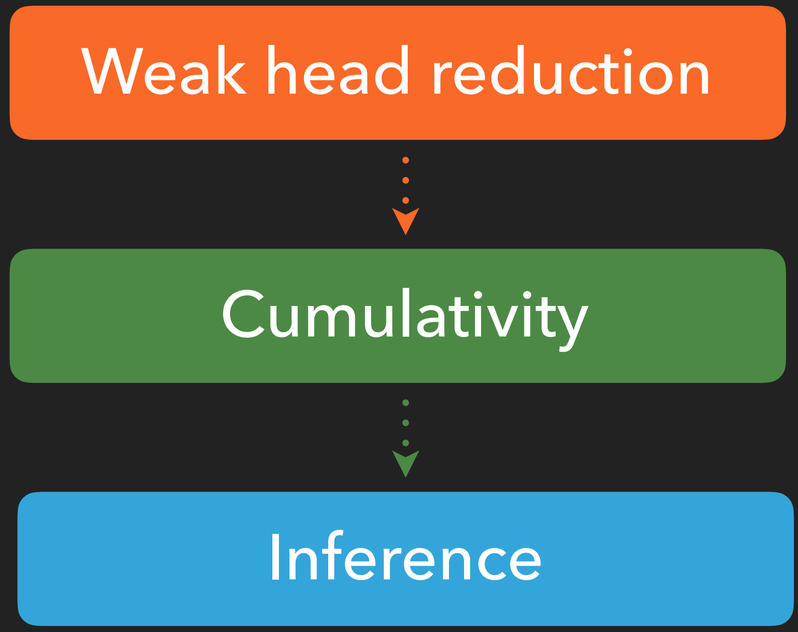


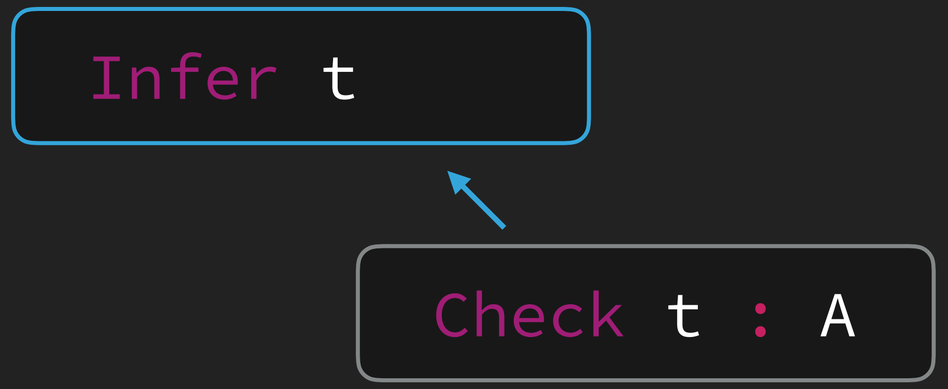
Type Checking



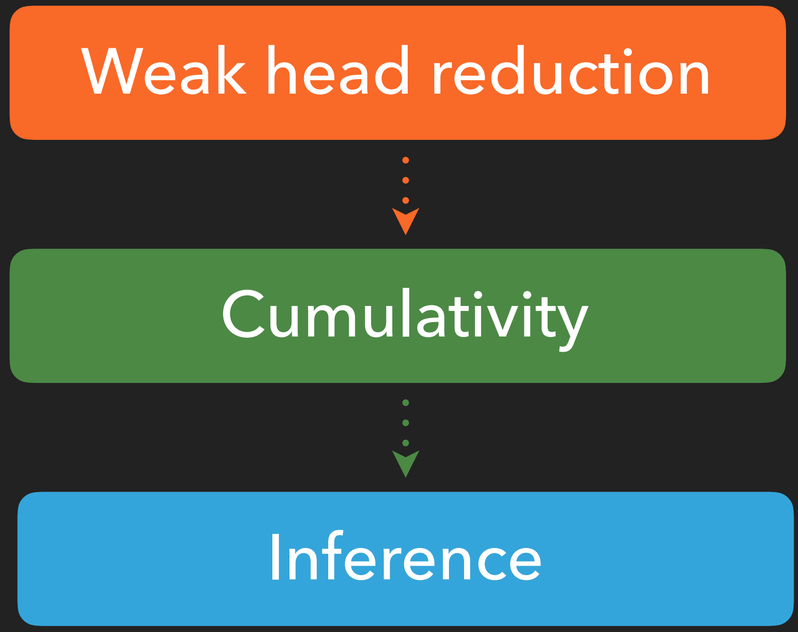

```
Check t : A
```

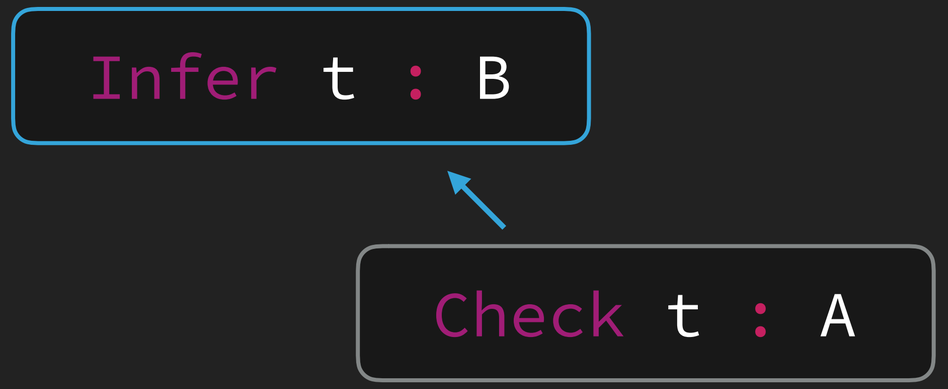
Type Checking



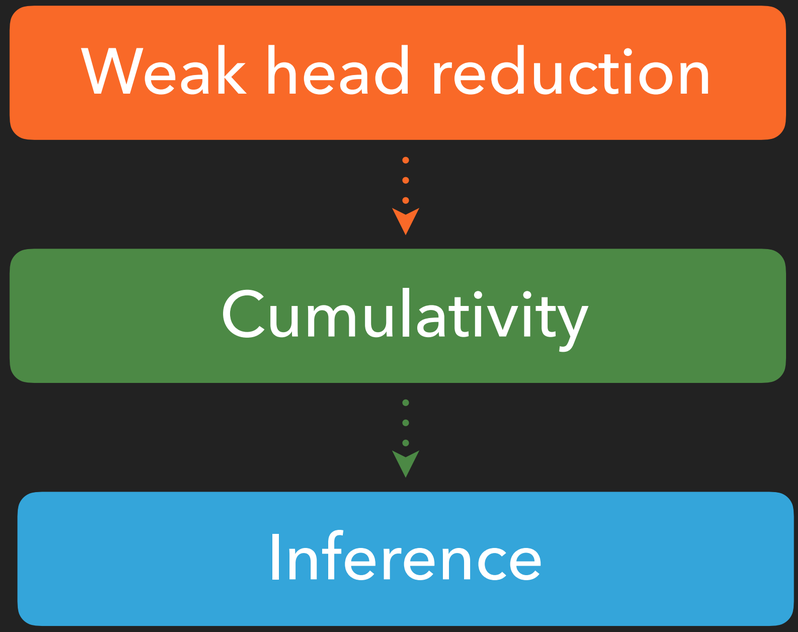


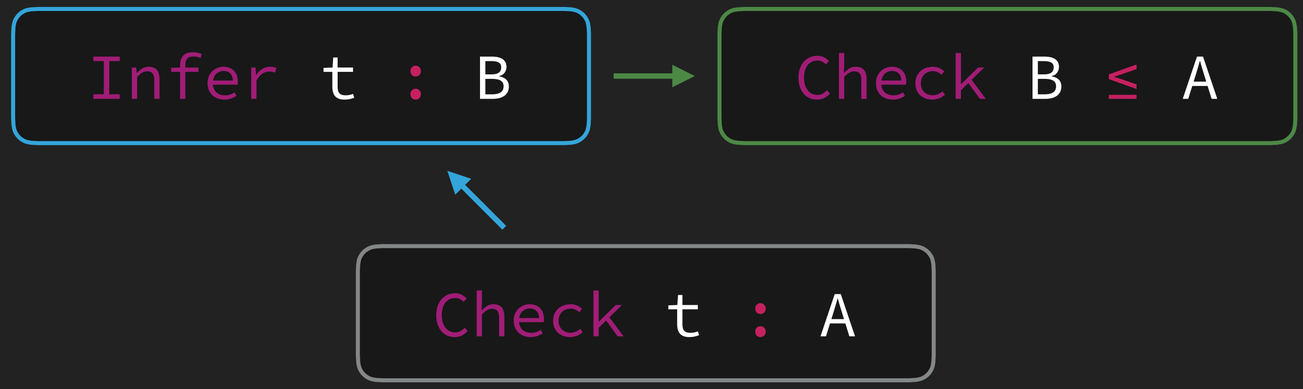
Type Checking



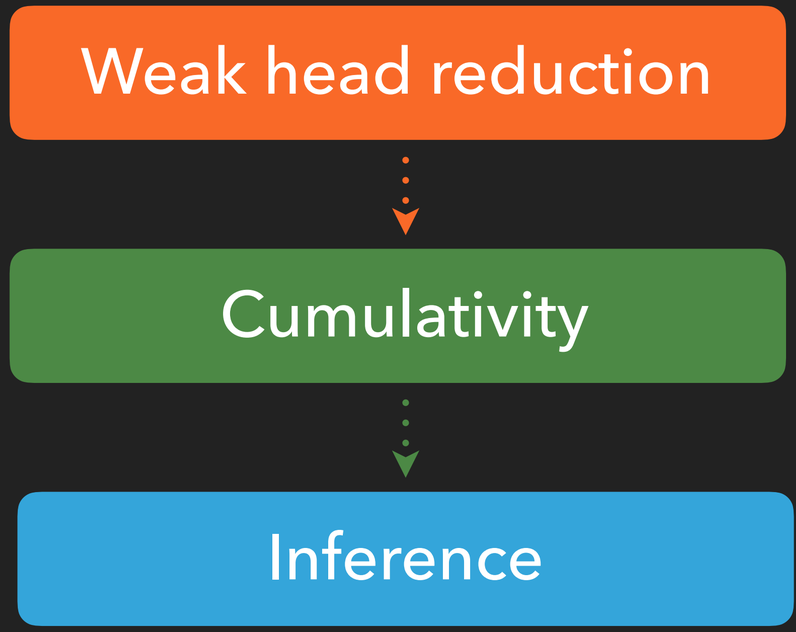


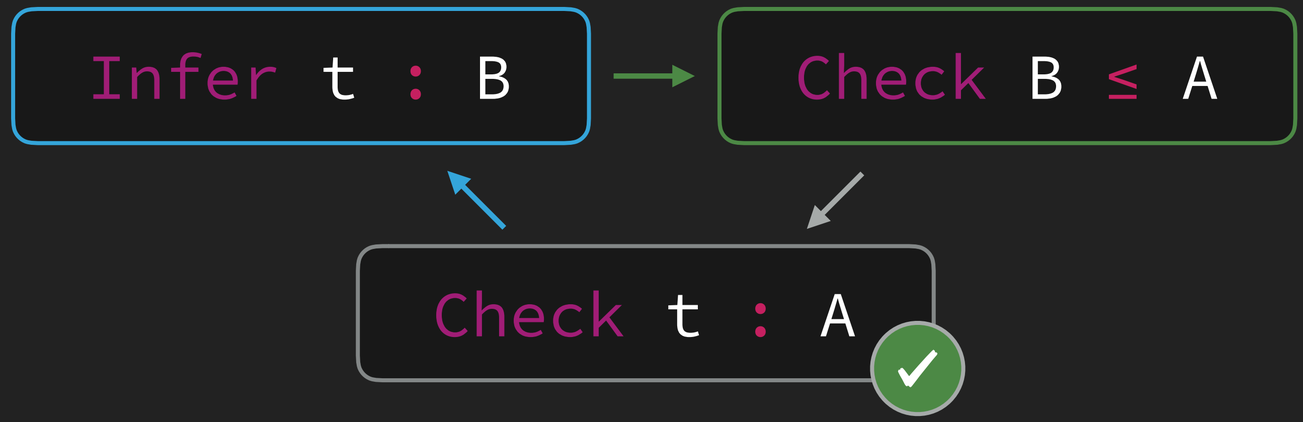
Type Checking





Type Checking





Verifying Erasure

Erasure

At the core of the **extraction** mechanism:

$\mathcal{E} : \text{term} \to \Lambda^{\square,\text{match,fix,cofix}}$

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)
```

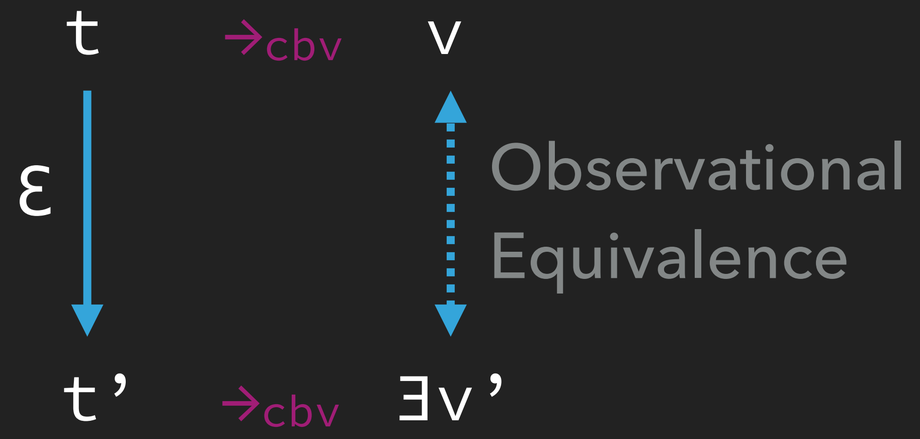
Ɛ (coerce) ~

```
coerce x y := (fun p ⇒ p)
```

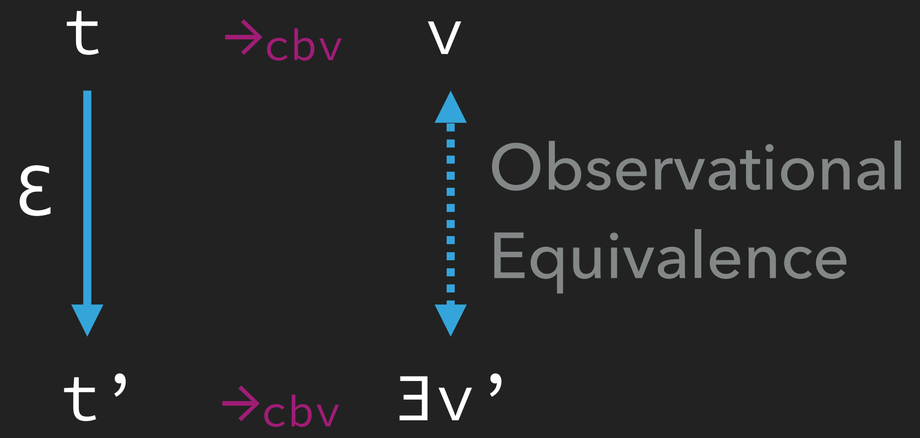
Ɛ (vrev) ~

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

Erasure Correctness



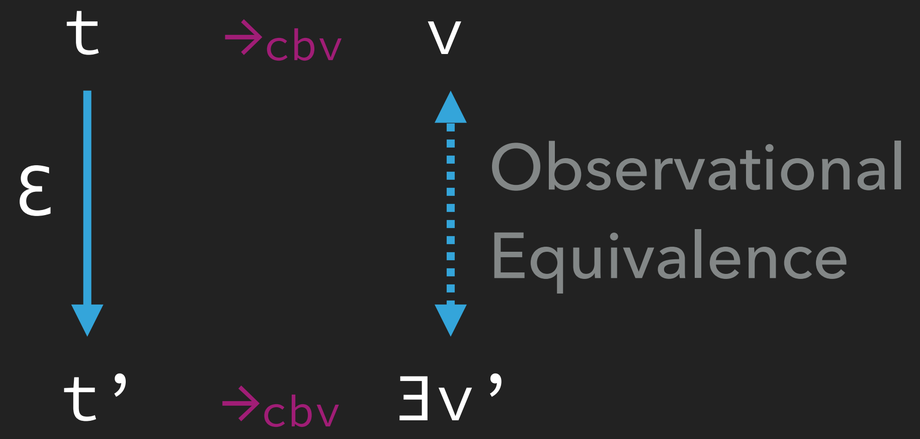
Erasure Correctness



With Canonicity and SN:

$$\vdash t : \mathsf{nat}$$

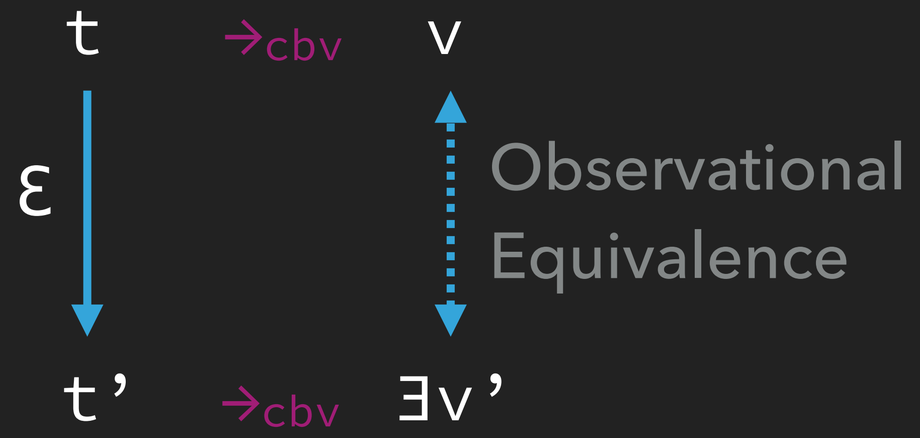
Erasure Correctness



With Canonicity and SN:

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

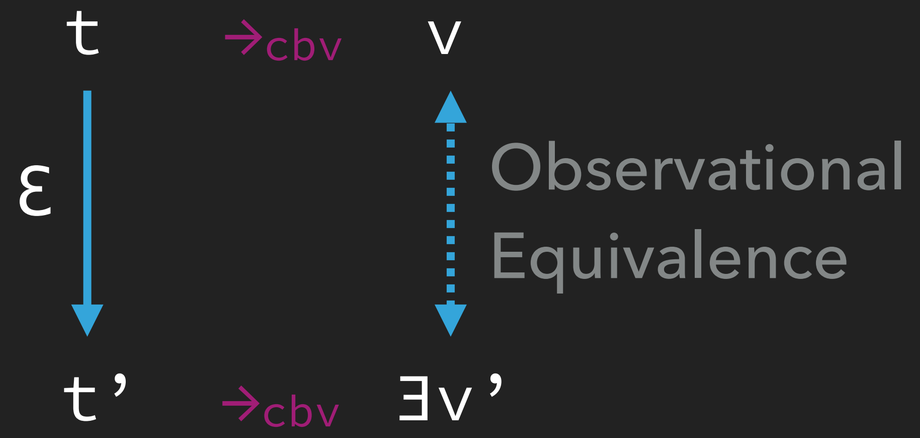
Erasure Correctness



With Canonicity and SN:

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

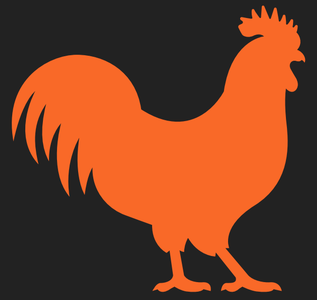
Erasure Correctness



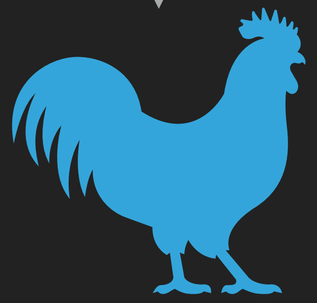
With Canonicity and SN:

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

Conclusion



Ideal Coq

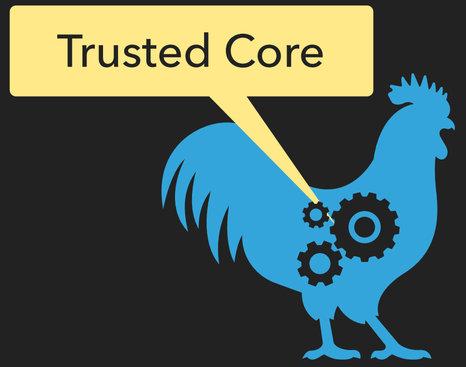


Verified Coq

in

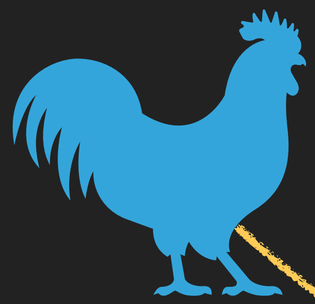
MetaCoq

in



Implemented Coq

Conclusion



Verified Coq

```
MetaCoq Check vrev.
```

Spec: 23kLoC

Proofs: 40kLoC

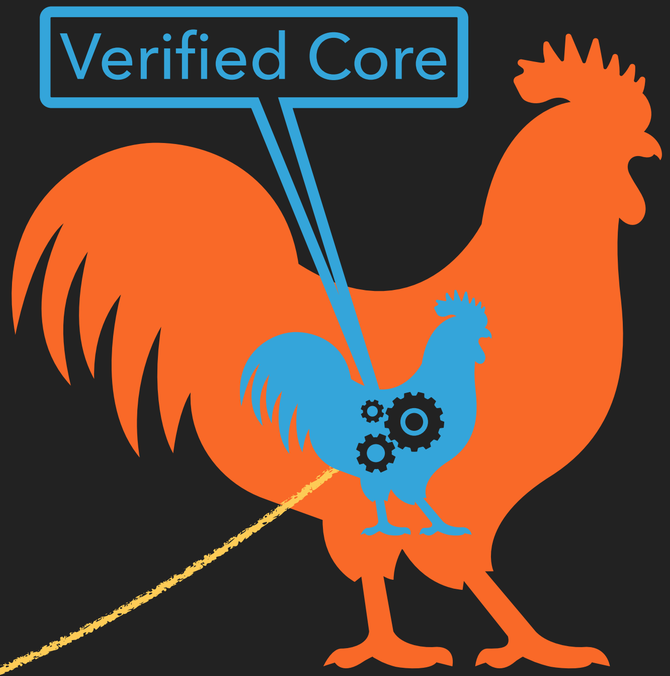
in

MetaCoq

Verified **ε**

```
MetaCoq Erase vrev.
```

in

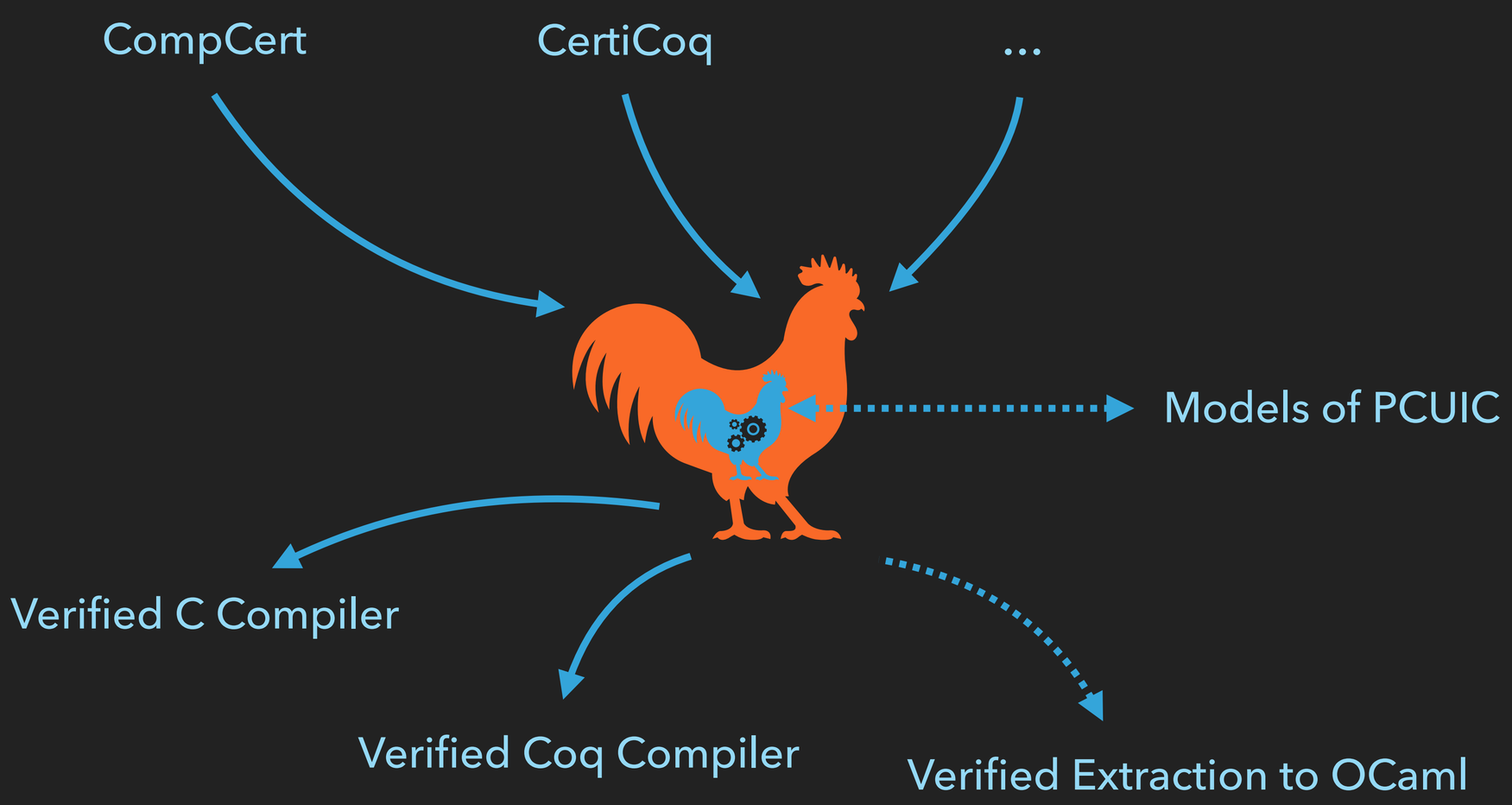


Implemented Coq

=

Ideal Coq

Perspectives



Conclusion

Spec: 23kLoC

Proofs: 40kLoC

https://metacoq.github.io/metacoq

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* with Coq (POPL'12)