EuroProofNet WG6 meeting 2024

Dependent Ghosts have a reflection for free

👻 + 🪞 = 🚫💲

Théo Winterhalter

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

actually a type mismatch!

`vec A (S k + m)` vs `vec A (k + S m)`

but we really wish they would be equal…

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

actually a type mismatch!

`vec A (S k + m)` vs `vec A (k + S m)`

but we really wish they would be equal…

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc)))
```

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

we should have lists!

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc))
```

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

we should have lists!

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc))
```

The problem is always in the n of `vec A n`...

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

erased is removed at extraction

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

erased is removed at extraction

Using ghost types...

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

erased is removed at extraction

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

```
gS : erased ℕ → erased ℕ
gS n := reveal n as x in hide (S x)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

erased is removed at extraction

Using ghost types...

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

erased is removed at extraction

```
gS : erased ℕ → erased ℕ
gS n := reveal n as x in hide (S x)
```

but...

```
erased bool → bool
```

only contains constant functions

…and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{\mathtt{A : Ghost} \qquad \mathtt{u, v : A} \qquad \mathtt{e : u = v}}{\mathtt{u \equiv v}}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

…and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

```
rev : ∀ {n m}. vec A n → vec A m → vec A (n +’ m)
rev vnil acc := acc
rev (vcons a k v) acc := rev v (vcons a m acc)
```

…and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

```
rev : ∀ {n m}. vec A n → vec A m → vec A (n +' m)
rev vnil acc := acc
rev (vcons a k v) acc := rev v (vcons a m acc)
```

ok because `vec A (gS k +' m) ≡ vec A (k +' gS m)`

Wait, couldn’t this just be (S)Prop?

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn’t this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

$$\frac{A : \mathtt{Prop} \qquad u, v : A}{u \equiv v}$$

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

$$\frac{\texttt{A : Prop} \qquad \texttt{u, v : A}}{\texttt{u} \equiv \texttt{v}}$$

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{\texttt{A : Ghost} \qquad \texttt{u, v : A} \qquad \texttt{e : u = v}}{\texttt{u} \equiv \texttt{v}}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)`

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

$$\frac{A : \mathtt{Prop} \qquad u, v : A}{u \equiv v}$$

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)`

and thus `hide 0` and `gS n`

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

$$\frac{\texttt{A : Prop} \qquad \texttt{u, v : A}}{\texttt{u} \equiv \texttt{v}}$$

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{\texttt{A : Ghost} \qquad \texttt{u, v : A} \qquad \texttt{e : u = v}}{\texttt{u} \equiv \texttt{v}}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

this is a problem though!

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)`

and thus `hide 0` and `gS n`

What is a safe reveal?

$$\frac{e : \texttt{erased}\ A \qquad P : \texttt{erased}\ A \to s \qquad f : \forall\ (x : A).\ P\ (\texttt{hide}\ x)}{\texttt{reveal}\ e\ P\ f : P\ e}$$

$s \in \{\ \texttt{Prop},\ \texttt{Ghost}\ \}$

`“reveal e as x in t” := reveal e P (λx. t)`

What is a safe reveal?

$$\frac{e : \mathtt{erased}\ A \qquad P : \mathtt{erased}\ A \to s \qquad f : \forall\ (x : A).\ P\ (\mathtt{hide}\ x)}{\mathtt{reveal}\ e\ P\ f : P\ e}$$

$s \in \{\ \mathtt{Prop},\ \mathtt{Ghost}\ \}$

`“reveal e as x in t” := reveal e P (λx. t)`

`reveal (hide t) P f ≡ f t`

What is a safe reveal?

```
e : erased A      P : erased A → s      f : ∀ (x : A). P (hide x)
─────────────────────────────────────────────────────────────────
                      reveal e P f : P e
```

`s ∈ { Prop, Ghost }`

`“reveal e as x in t” := reveal e P (λx. t)`

`reveal (hide t) P f ≡ f t`

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

What is a safe reveal?

$$\frac{\mathtt{e : erased\ A} \qquad \mathtt{P : erased\ A \to s} \qquad \mathtt{f : \forall\ (x : A).\ P\ (hide\ x)}}{\mathtt{reveal\ e\ P\ f : P\ e}}$$

`s ∈ { Prop, Ghost }`

`“reveal e as x in t” := reveal e P (λx. t)`

`reveal (hide t) P f ≡ f t`

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

```
D : erased ℕ → Prop
D n := reveal n as x in match x with 0 => ⊤ | _ => ⊥ end
```

What is a safe reveal?

```
e : erased A      P : erased A → s      f : ∀ (x : A). P (hide x)
─────────────────────────────────────────────────────────────────
                      reveal e P f : P e
```

`s ∈ { Prop, Ghost }`

`“reveal e as x in t” := reveal e P (λx. t)`

`reveal (hide t) P f ≡ f t`

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

```
D : erased ℕ → Prop
D n := reveal n as x in match x with 0 => ⊤ | _ => ⊥ end
```

But here `P _ := Prop` so `P : erased ℕ → Type`

What is a safe reveal?

$$\frac{e : \mathtt{erased}\ A \qquad P : \mathtt{erased}\ A \to s \qquad f : \forall\ (x : A).\ P\ (\mathtt{hide}\ x)}{\mathtt{reveal}\ e\ P\ f : P\ e}$$

$s \in \{ \mathtt{Prop}, \mathtt{Ghost} \}$

“`reveal e as x in t`” := `reveal e P (λx. t)`

`reveal (hide t) P f ≡ f t`

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

```
D : erased ℕ → Prop
D n := reveal n as x in match x with 0 => ⊤ | _ => ⊥ end
```

But here `P _ := Prop` so `P : erased ℕ → Type`

Reveal proposition

$$\frac{\texttt{e : erased A} \qquad \texttt{f : A → Prop}}{\texttt{Reveal e f : Prop}}$$

`Reveal (hide t) f ↔ f t` no computation

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

```
D : erased ℕ → Prop
D n := reveal n as x in match x with 0 => ⊤ | _ => ⊥ end
```

But here `P _ := Prop` so `P : erased ℕ → Type`

Reveal proposition

```
e : erased A      f : A → Prop
──────────────────────────────
       Reveal e f : Prop
```

`Reveal (hide t) f ↔ f t` no computation

We want a discriminator: `D (hide 0) ≡ ⊤` `D (gS n) ≡ ⊥`

```
D : erased ℕ → Prop
D n := Reveal n (λx. match x with 0 => ⊤ | _ => ⊥ end)
```

Reveal proposition

$$\frac{e : \mathtt{erased}\ A \qquad f : A \to \mathtt{Prop}}{\mathtt{Reveal}\ e\ f : \mathtt{Prop}}$$

`Reveal (hide t) f ↔ f t` no computation

We get a discriminator: `D (hide 0) ↔ ⊤` `D (gS n) ↔ ⊥`

```
D : erased ℕ → Prop
D n := Reveal n (λx. match x with 0 => ⊤ | _ => ⊥ end)
```

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

$$\Big\downarrow$$

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

How do we justify this?

$$\frac{\texttt{A : Ghost} \qquad \texttt{u, v : A} \qquad \texttt{e : u = v}}{\texttt{u} \equiv \texttt{v}}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

$$\frac{\texttt{A : Ghost} \qquad \texttt{e : u} =_{\texttt{A}} \texttt{v} \qquad \texttt{t : P u}}{\texttt{cast e P t : P v}}$$

Ghost casts

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

MLTT/CIC with (S)Prop

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

parametricity translation
taking inspiration from exceptional type theory
[Pédrot Tabareau 2018]

MLTT/CIC with (S)Prop

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

GTT

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

interesting on its own!

parametricity translation
taking inspiration from exceptional type theory
[Pédrot Tabareau 2018]

MLTT/CIC with (S)Prop

Erasure

Idea: Getting rid of all ghosts

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{A} : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [\mathtt{t}]_\varepsilon : [\![\mathtt{A}]\!]_\varepsilon$ in CC

Erasure

Idea: Getting rid of all ghosts

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

Erasure

Idea: Getting rid of all ghosts

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{A} : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [\mathtt{t}]_\varepsilon : [\![\mathtt{A}]\!]_\varepsilon$ in CC

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{A} : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[\mathtt{t}]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A :$ `Type` in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A :$ `Ghost`/`Prop` in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A :$ `Type` in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A :$ `Ghost`/`Prop` in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathsf{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathsf{Ghost}/\mathsf{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

$[\mathsf{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$$[x^{\mathbb{T}}]_\varepsilon := x$$

$$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$$

$$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$$

$$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$$[x^{\mathbb{T}}]_\varepsilon := x$$

$$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$$

$$[\lambda (x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda (x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$$

$$[\lambda (x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$$

$$[\mathtt{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := ??$$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash \mathtt{t} :: \mathbb{T}$

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{A} : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [\mathtt{t}]_\varepsilon : [\![\mathtt{A}]\!]_\varepsilon$ in CC

or $\Gamma \vdash \mathtt{t} :: \mathbb{G}$ / $\Gamma \vdash \mathtt{t} :: \mathbb{P}$

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{A} : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[\mathtt{t}]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[\mathtt{x}^{\mathbb{T}}]_\varepsilon := \mathtt{x}$

$[\mathtt{cast}\ \mathtt{e}\ \mathtt{P}\ \mathtt{t}]_\varepsilon := [\mathtt{t}]_\varepsilon$

$[\lambda(\mathtt{x}^{\mathbb{T}} : \mathtt{A}).\ \mathtt{t}]_\varepsilon := \lambda(\mathtt{x} : [\![\mathtt{A}]\!]_\varepsilon).\ [\mathtt{t}]_\varepsilon$

$[\lambda(\mathtt{x}^{\mathbb{G}} : \mathtt{A}).\ \mathtt{t}]_\varepsilon := [\mathtt{t}]_\varepsilon$

$[\mathtt{f}^{\mathbb{T}}\ \mathtt{u}^{\mathbb{T}}]_\varepsilon := [\mathtt{f}]_\varepsilon\ [\mathtt{u}]_\varepsilon$

$[\mathtt{f}^{\mathbb{T}}\ \mathtt{u}^{\mathbb{G}}]_\varepsilon := [\mathtt{f}]_\varepsilon$

$\mathtt{exfalso}^{\mathbb{T}}\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{p} : \bot) : \mathtt{A}$

$[\mathtt{exfalso}^{\mathbb{T}}\ \mathtt{A}\ \mathtt{p}]_\varepsilon := \mathtt{??}$

Erasure

Idea: Getting rid of all ghosts

or $\Gamma \vdash t :: \mathbb{T}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Type}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

or $\Gamma \vdash t :: \mathbb{G}$ / $\Gamma \vdash t :: \mathbb{P}$

If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \mathtt{Ghost}/\mathtt{Prop}$ in GTT
then $[t]_\varepsilon$ is undefined

We in fact have modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$[x^{\mathbb{T}}]_\varepsilon := x$

$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

$\mathtt{exfalso}^{\mathbb{T}}\ (A : \mathtt{Type})\ (p : \bot) : A$

$[\mathtt{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := ??$

we get no $\bot$ but we need some $[\![A]\!]_\varepsilon$

Erasure

Exceptionally [Pédrot Tabareau 2018]

`exfalso`$^{\mathbb{T}}$ `(A : Type) (p : ⊥) : A`

`[exfalso`$^{\mathbb{T}}$ `A p]`$_{\varepsilon}$ `:=` “`raise` `⟦A⟧`$_{\varepsilon}$”

like `assert false` in OCaml

Erasure

Exceptionally [Pédrot Tabareau 2018]

$\mathtt{exfalso}^{\mathbb{T}}\ \mathtt{(A : Type)\ (p : \bot) : A}$

$\mathtt{[exfalso}^{\mathbb{T}}\ \mathtt{A\ p]}_{\varepsilon}\ \mathtt{:=}$ “$\mathtt{raise}\ [\![\mathtt{A}]\!]_{\varepsilon}$”

like `assert false` in OCaml

$$\mathtt{A : Type} \longrightarrow [\![\mathtt{A}]\!]_{\varepsilon}\ \mathtt{: Type}$$

Erasure

Exceptionally [Pédrot Tabareau 2018]

$\texttt{exfalso}^{\mathbb{T}}\ \texttt{(A : Type) (p : }\bot\texttt{) : A}$

$[\texttt{exfalso}^{\mathbb{T}}\ \texttt{A p}]_{\varepsilon}$:= “raise $[\![\texttt{A}]\!]_{\varepsilon}$”

like `assert false` in OCaml

$$\texttt{A : Type} \longrightarrow [\![\texttt{A}]\!]_{\varepsilon}\ \texttt{: Type} \qquad [\![\texttt{A}]\!]_{\varnothing}\ \texttt{:}\ [\![\texttt{A}]\!]_{\varepsilon}$$

Erasure

Exceptionally [Pédrot Tabareau 2018]

$\mathtt{exfalso}^{\mathbb{T}}\ (A : \mathtt{Type})\ (p : \bot) : A$

$[\mathtt{exfalso}^{\mathbb{T}}\ A\ p]_{\varepsilon} := [\![A]\!]_{\varnothing}$

like `assert false` in OCaml

$$A : \mathtt{Type} \longrightarrow [\![A]\!]_{\varepsilon} : \mathtt{Type} \qquad [\![A]\!]_{\varnothing} : [\![A]\!]_{\varepsilon}$$

Erasure

Exceptionally [Pédrot Tabareau 2018]

$\mathtt{exfalso}^{\mathbb{T}}\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{p} : \bot) : \mathtt{A}$

$[\mathtt{exfalso}^{\mathbb{T}}\ \mathtt{A}\ \mathtt{p}]_{\varepsilon} := [\![\mathtt{A}]\!]_{\varnothing}$

like `assert false` in OCaml

$$\mathtt{A} : \mathtt{Type} \longrightarrow [\![\mathtt{A}]\!]_{\varepsilon} : \mathtt{Type} \qquad [\![\mathtt{A}]\!]_{\varnothing} : [\![\mathtt{A}]\!]_{\varepsilon}$$

$[\mathtt{Type}]_{\varepsilon} := \mathtt{ty}$

```
Inductive ty : Type :=
| tyval (A : Type) (a : A)
| tyerr
```

Erasure

Exceptionally [Pédrot Tabareau 2018]

```
exfalso𝕋 (A : Type) (p : ⊥) : A
```

```
[exfalso𝕋 A p]ε := ⟦A⟧∅
```

like `assert false` in OCaml

```
A : Type ⟶ ⟦A⟧ε : Type    ⟦A⟧∅ : ⟦A⟧ε
```

```
[Type]ε := ty
```

```
Inductive ty : Type :=
| tyval (A : Type) (a : A)
| tyerr
```

```
El (tyval A a) ≡ A
```

```
Err (tyval A a) ≡ a
```

```
El tyerr ≡ unit
```

```
Err tyerr ≡ ()
```

Erasure

Exceptionally [Pédrot Tabareau 2018]

```
exfalso⊤ (A : Type) (p : ⊥) : A
```

```
[exfalso⊤ A p]ε := ⟦A⟧∅
```

like `assert false` in OCaml

```
A : Type  ⟶  ⟦A⟧ε : Type    ⟦A⟧∅ : ⟦A⟧ε
```

```
[Type]ε := ty
```

```
Inductive ty : Type :=
| tyval (A : Type) (a : A)
| tyerr
```

```
El (tyval A a) ≡ A
```

```
Err (tyval A a) ≡ a
```

```
El tyerr ≡ unit
```

```
Err tyerr ≡ ()
```

```
⟦A⟧ε := El [A]ε
```

```
⟦A⟧∅ := Err [A]ε
```

Parametricty

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{t} :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_{\mathrm{P}} \vdash [\![\mathtt{t}]\!]_{\mathrm{P}} : [\![\mathtt{A}]\!]_{\mathrm{P}}\ [\mathtt{t}]_{\varepsilon}$ in CC

Parametricty

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{t} :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_{\mathtt{P}} \vdash [\![\mathtt{t}]\!]_{\mathtt{P}} : [\![\mathtt{A}]\!]_{\mathtt{P}}\ [\mathtt{t}]_{\varepsilon}$ in CC

Proposition guaranteeing no exceptions raised at top-level

Parametricty

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{t} :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_{P} \vdash [\![\mathtt{t}]\!]_{P} : [\![\mathtt{A}]\!]_{P}\ [\mathtt{t}]_{\varepsilon}$ in CC

Proposition guaranteeing no exceptions raised at top-level
Parametricity in `Prop` [Keller Lasson 2012]

Parametricty

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{t} :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![\mathtt{t}]\!]_P : [\![\mathtt{A}]\!]_P\ [\mathtt{t}]_\varepsilon$ in CC

Proposition guaranteeing no exceptions raised at top-level
Parametricity in `Prop` [Keller Lasson 2012]
⇒ Limitations: no large elimination! 😔

Parametricty

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [t]_\varepsilon$ in CC

Proposition guaranteeing no exceptions raised at top-level
Parametricity in `Prop` [Keller Lasson 2012]
⇒ Limitations: no large elimination! 😔

still ok for small inductives (like bool)

Parametricty

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [t]_\varepsilon$ in CC

Proposition guaranteeing no exceptions raised at top-level
Parametricity in `Prop` [Keller Lasson 2012]
⇒ Limitations: no large elimination! 😔

still ok for small inductives (like bool)

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{P}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P$ in CC

Propositions essentially preserved (including $\bot$)
Hence consistency

Parametricty

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [t]_\varepsilon$ in CC

Proposition guaranteeing no exceptions raised at top-level
Parametricity in `Prop` [Keller Lasson 2012]
⇒ Limitations: no large elimination! 😔

still ok for small inductives (like bool)

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [\![t]\!]_v$ in CC

Ghost values are erased: we need to revive them

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{P}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P$ in CC

Propositions essentially preserved (including $\perp$)
Hence consistency

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash \mathtt{t} : \mathtt{A}$ and $\Gamma \vdash \mathtt{t} :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![\mathtt{t}]\!]_v : [\![\mathtt{A}]\!]_\varepsilon$ in CC

(undefined otherwise)

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

$$[\![\texttt{hide}\ t]\!]_v := [t]_\varepsilon$$

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

$[\![\texttt{hide}\ t]\!]_v := [t]_\varepsilon$

$[\![\texttt{reveal}\ e\ P\ f^{\mathbb{G}}]\!]_v := [\![f]\!]_v\ [\![e]\!]_v$

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

$[\![\texttt{hide}\ t]\!]_v := [t]_\varepsilon$

$[\![\texttt{reveal}\ e\ P\ f^{\mathbb{G}}]\!]_v := [\![f]\!]_v\ [\![e]\!]_v$

$[\![f^{\mathbb{G}}\ u^{\mathbb{T}}]\!]_v := [\![f]\!]_v\ [u]_\varepsilon$

$[\![f^{\mathbb{G}}\ u^{\mathbb{G}}]\!]_v := [\![f]\!]_v\ [\![u]\!]_v$

$[\![f^{\mathbb{G}}\ u^{\mathbb{P}}]\!]_v := [\![f]\!]_v$

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolₚ : bool• → Prop :=
| trueₚ : boolₚ true•
| falseₚ : boolₚ false•
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolₚ : bool• → Prop :=
| trueₚ : boolₚ true•
| falseₚ : boolₚ false•
```

Free theorem:

```
erased bool → bool
```

only contains constant functions

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolₚ : bool• → Prop :=
| trueₚ : boolₚ true•
| falseₚ : boolₚ false•
```

Free theorem:

```
erased bool → bool
```

only contains constant functions

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolₚ : bool• → Prop :=
| trueₚ : boolₚ true•
| falseₚ : boolₚ false•
```

Free theorem:

```
erased bool → bool
```

only contains constant functions

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolₚ b → boolₚ fe)
  (P : bool• → unit) (PP : ∀ b, boolₚ b → Prop),
  PP fe (fP true• trueₚ) → PP fe (fP false• falseₚ)
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolₚ : bool• → Prop :=
| trueₚ : boolₚ true•
| falseₚ : boolₚ false•
```

Free theorem:

```
erased bool → bool
```

only contains constant functions

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolₚ b → boolₚ fe)
  (P : bool• → unit) (PP : ∀ b, boolₚ b → Prop),
  PP fe (fP true• trueₚ) → PP fe (fP false• falseₚ)
```

proof irrelevance

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity

```
Inductive boolP : bool• → Prop :=
| trueP : boolP true•
| falseP : boolP false•
```

Free theorem:

```
erased bool → bool
```

only contains constant functions

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolP b → boolP fe)
  (P : bool• → unit) (PP : ∀ b, boolP b → Prop),
  PP fe (fP true• trueP) → PP fe (fP false• falseP)
```

the key is that it is erased to a constant

proof irrelevance

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

```
Inductive vecₚ (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕₚ n), vec• A → Prop :=
| vnilₚ : vecₚ A AP 0• 0ₚ vnil•
| vconsₚ a (aP : AP a) n nP v : vecₚ A AP n nP v → vecₚ A AP (S• n) (Sₚ n nP) (vcons• a v)
```

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

```
Inductive vecₚ (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕₚ n), vec• A → Prop :=
| vnilₚ : vecₚ A AP 0• 0ₚ vnil•
| vconsₚ a (aP : AP a) n nP v : vecₚ A AP n nP v → vecₚ A AP (S• n) (Sₚ n nP) (vcons• a v)
```

```
⟦vec A n⟧ₚ := vecₚ ⟦A⟧ε ⟦A⟧ₚ ⟦n⟧ᵥ ⟦n⟧ₚ
```

first real use of revival

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

```
Inductive vecₚ (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕₚ n), vec• A → Prop :=
| vnilₚ : vecₚ A AP 0• 0ₚ vnil•
| vconsₚ a (aP : AP a) n nP v : vecₚ A AP n nP v → vecₚ A AP (S• n) (Sₚ n nP) (vcons• a v)
```

```
⟦vec A n⟧ₚ := vecₚ [A]ε ⟦A⟧ₚ ⟦n⟧ᵥ ⟦n⟧ₚ
```

first real use of revival

Computation for the eliminator

```
vec-elim vnil P z s ≡ z
```

Unusual

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec⌀
```

```
Inductive vecₚ (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕₚ n), vec• A → Prop :=
| vnilₚ : vecₚ A AP 0• 0ₚ vnil•
| vconsₚ a (aP : AP a) n nP v : vecₚ A AP n nP v → vecₚ A AP (S• n) (Sₚ n nP) (vcons• a v)
```

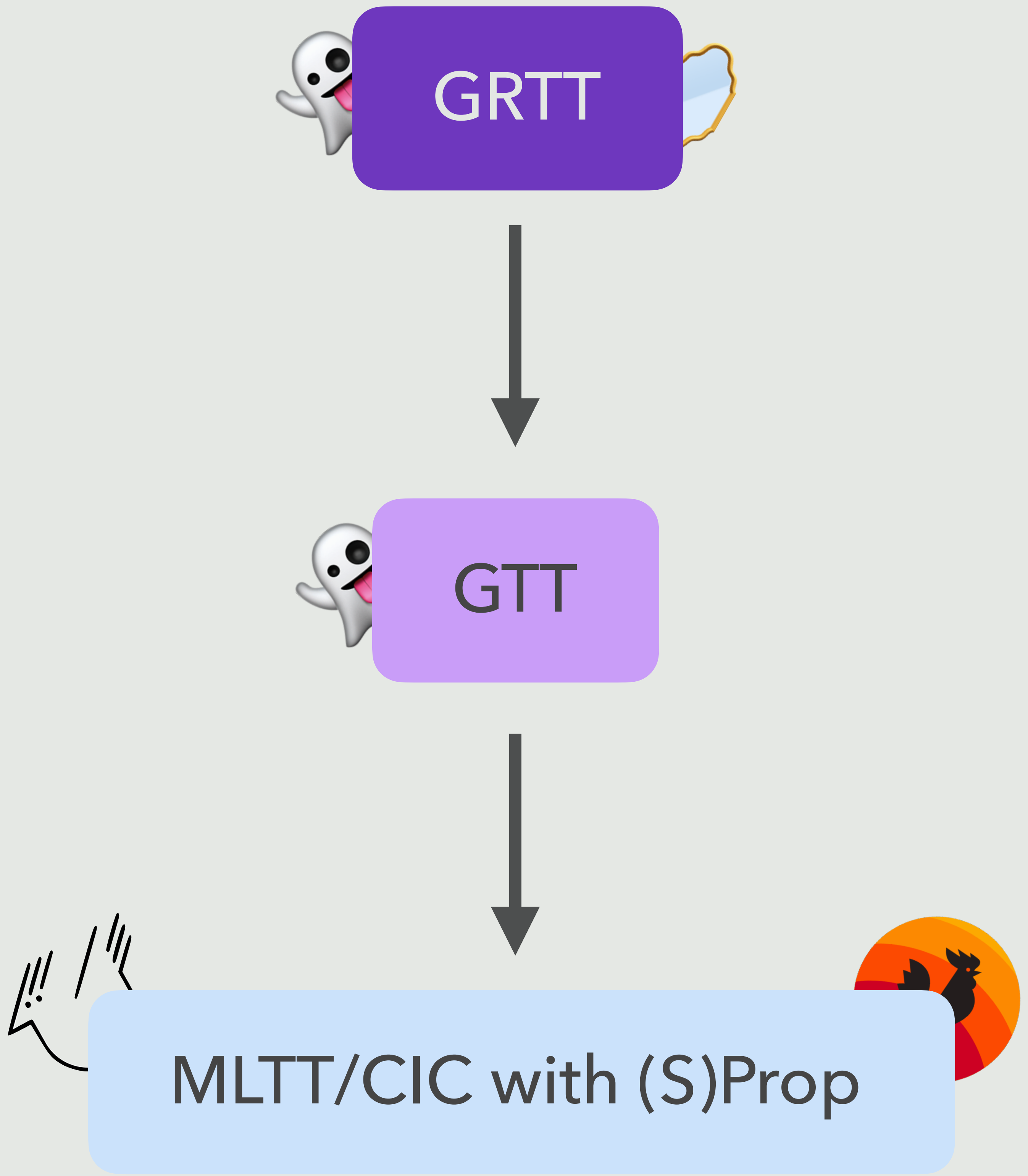
```
⟦vec A n⟧ₚ := vecₚ [A]ε ⟦A⟧ₚ ⟦n⟧ᵥ ⟦n⟧ₚ
```

first real use of revival

Computation for the eliminator

```
vec-elim vnil P z s ≡ z
```

```
vec-elim (vcons a n v) P z s ≡ s a (glength v) v (vec-elim v P z s)
```



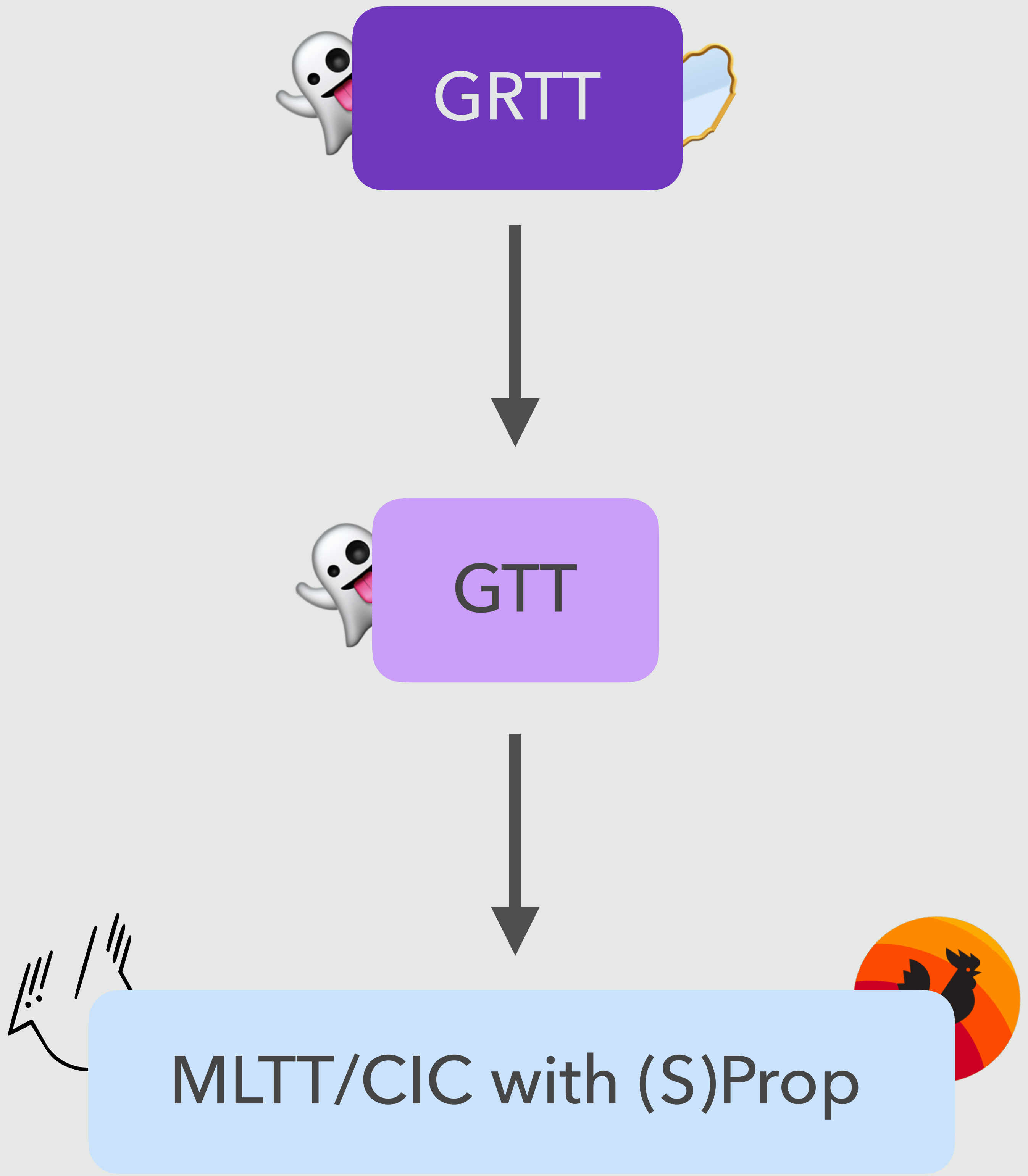
GRTT

GTT

MLTT/CIC with (S)Prop

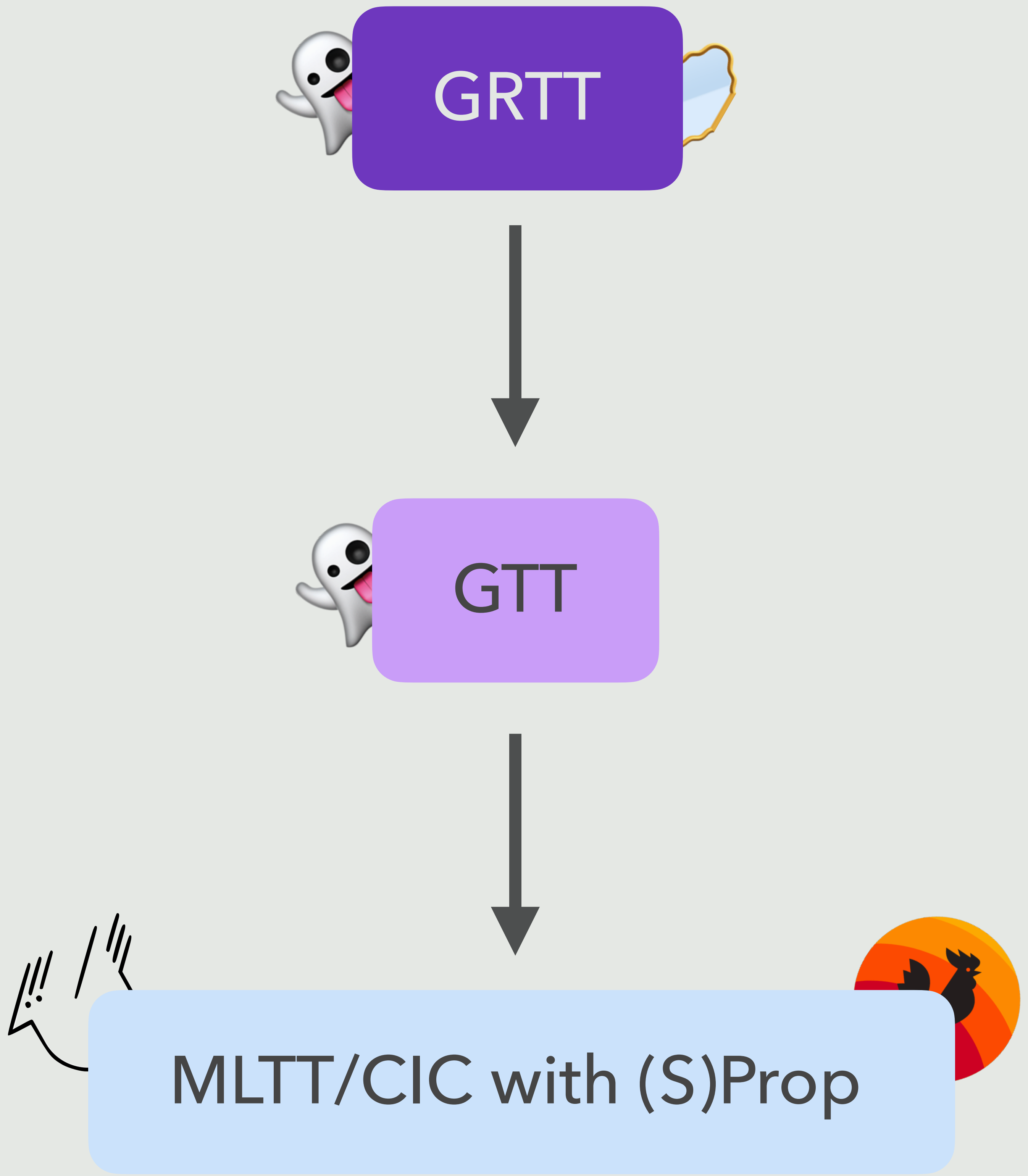
Meta-theory

conservativity
consistency
type former discrimination
free theorems



Meta-theory

conservativity
consistency
type former discrimination
free theorems



GRTT

GTT

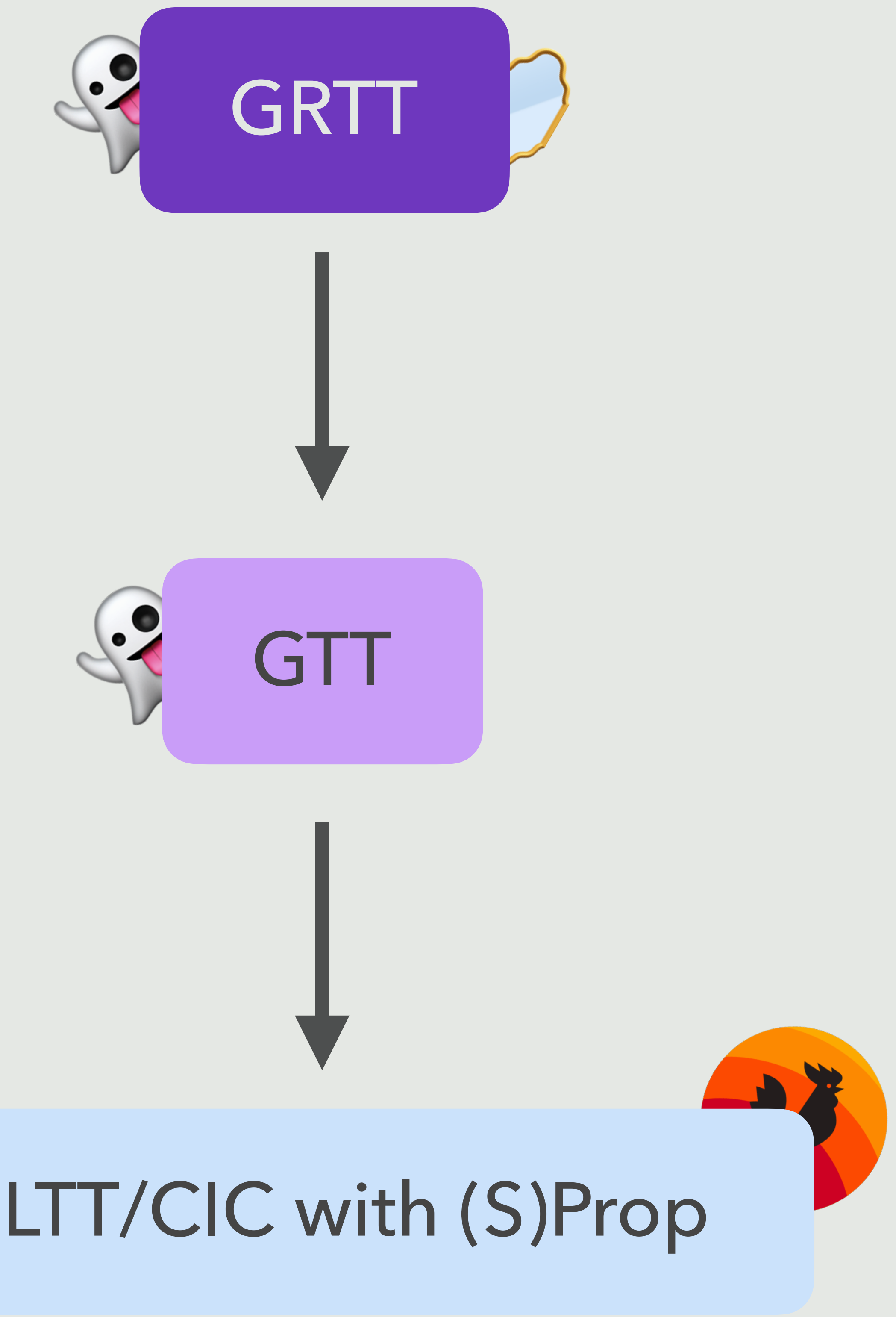
MLTT/CIC with (S)Prop

Perspectives

general inductives
subject reduction
termination
decidability (for GTT only)
meta-theory of F*

Meta-theory

conservativity
consistency
type former discrimination
free theorems



GRTT

GTT

MLTT/CIC with (S)Prop

Perspectives

general inductives
subject reduction
termination
decidability (for GTT only)
meta-theory of F*

some tricks in the formalisation
to handle contexts of varying size

Autosubst 2 is very useful
but automation pushed to the limits

/TheoWinterhalter/ghost-reflection